Snap-stabilizing Optimal Binary Search Tree

Doina Bein¹, Ajoy K. Datta¹, and Vincent Villain²

¹ School of Computer Science, University of Nevada, Las Vegas, 
{siona,datta}@cs.unlv.edu
² LaRIA, Université de Picardie Jules Verne, France, 
villain@laria.u-picardie.fr

Abstract. We present the first snap-stabilizing distributed binary search tree (BST) algorithm. A snap-stabilizing algorithm guarantees that the system always behaves according to its specification provided some processor initiated the protocol. The maximum number of items that can be stored at any time at any processor is constant (independent of the size (n) of the network). Under this space constraint, we show a lower bound of $\Omega(n)$ on the time complexity for the BST problem. We then prove that starting from an arbitrary configuration where the nodes have distinct internal values drawn from an arbitrary set, our algorithm arranges them in a BST order in $O(n)$ rounds. Therefore, our solution is asymptotically optimal in time and takes $O(n)$ rounds. A processor $i$ requires $O(\log s_i)$ bits of space where $s_i$ is the size of the subtree rooted at $i$. So, the root uses $O(\log n)$ bits. The proposed algorithm uses a heap algorithm as a preprocessing step. This is also the first snap-stabilizing distributed solution to the heap problem. The heap construction spends $O(h)$ (where $h$ is the height of the tree) rounds. Its space requirement is constant (independent of $n$). We then exploit the heap in the next phase of the protocol. The root collects values in decreasing order and delivers them to each node in the tree in $O(n)$ rounds following a pipelined delivery order of sorted values in decreasing order.

Keywords: Binary search tree, heap, self-stabilization, snap-stabilization.

1 Introduction

Given a binary tree where every node holds one key (value) drawn from an arbitrary set of real values, we design a snap-stabilizing distributed algorithm to arrange the values in the tree to obtain a binary search tree. A self-stabilizing [5, 6] system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. A snap-stabilizing [2, 4] algorithm guarantees that it always behaves according to its specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps.

The BST construction works as follows. First, the values in the tree are re-arranged as a heap (we implement a MaxHeap but a MinHeap is equally possible). Based on the heap arrangement, the root collects values in decreasing order and delivers them to each node in the tree (a sequential, pipelined delivery
of sorted values in decreasing order). The tree structure is not modified by our algorithm.

Related Work: A heap construction that supports insert and delete operations in arbitrary states over a variant of the standard binary heap [3] with the maximum capacity of $K$ items is proposed in [8]. It takes $O(m \log K)$ heap operations to stabilize ($m$ is the initial number of items in the heap). The space complexity per node $i$ is $O(h_i)$ where $h_i$ is the height of the subtree $T_i$ in the binary heap rooted at node $i$. Stabilizing search 2-3 trees are investigated in [9]. The stabilization time is $O(n \log n)$ rounds where $n$ is the number of nodes in the initial state and the space complexity per node $i$ is $O(d_i)$ where $d_i$ is the distance from the root to node $i$.

Contributions: This paper has two major contributions. It includes the first snap-stabilizing binary search tree (BST) and the first snap-stabilizing heap algorithm. Being snap-stabilizing gives our algorithms a unique feature — they always behave as expected by their specifications. It should be noted that a self-stabilizing algorithm is guaranteed to satisfy the desired specification only in a finite time. In the context of the BST problem, in a self-stabilizing BST solution, if the root initiates a BST computation, it is not guaranteed that the tree will become a BST when the computation terminates. If the computation is repeated (a bounded but unknown number of times), the self-stabilizing algorithm guarantees that eventually, the tree will become a BST. The proposed snap-stabilizing solution achieves a much better solution than the above. It ensures that when a BST computation initiated by the root terminates, the tree is a BST. Thus, we do not need to repeat the computation unless the application program demands repeated sorting of the values in the tree.

A key feature of our solution is that the maximum number of items that can be stored at any time at any processor is constant (independent of the size ($n$) of the network). Under this space constraint, our solution is asymptotically optimal in time and takes $O(n)$ rounds. A processor $i$ requires $O(\log s_i)$ bits where $s_i$ is the size of the subtree rooted at $i$. So, the root uses $O(\log n)$ bits. The proposed algorithm uses a snap-stabilizing heap algorithm as a preprocessing step. This is also the first snap-stabilizing distributed solution to the heap problem. The cost of the heap construction is $O(h)$ rounds and constant (independent of $n$) space.

Outline of the paper: In Section 2, we present the computational model, snap-stabilization, and the specification of the BST problem. We present the solution (the detail code of the algorithm) in Section 3. Due to lack of space, the detail code of the predicates and macros are omitted. They are available in the technical report [1]. We give a sketch of the correctness proof in Section 4, while the detail proof is available in [1]. We finish the paper with some concluding remarks in Section 5.

2 Preliminaries

Distributed System: We consider an asynchronous binary tree network of $n$ processors with distinct ID’s. The root is denoted by $r$. We will use nodes and pro-
cessors interchangeably. The processors communicate using bi-directional links. We assume the local shared memory model of communication. The program of every processor consists of a set of shared variables and a finite set of actions. A processor can only write to its own variables, and read its own variables and variables owned by the neighboring processors. Each action is of the following form: \(<label> <guard> \rightarrow <statement>\). The guard of an action in the program of any process \(p\) is a boolean expression involving the variables of \(p\) and its neighbors. The statement of an action of \(p\) updates one or more variables of \(p\). An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The state of a processor is defined by the value of its variables. The state of a system is the product of the states of all processors. We will refer to the state of a processor and system as a (local) state and (global) configuration, respectively. A processor \(p\) is said to be enabled in a configuration \(\gamma\) if there exists at least an action \(A\) such that the guard of \(A\) is true in \(\gamma\). We consider that any processor \(p\) executed a disabling action in the computation step \(\gamma_i \rightarrow \gamma_{i+1}\) if \(p\) was enabled in \(\gamma_i\) and not enabled in \(\gamma_{i+1}\), but did not execute any action between these two configurations. (The disabling action represents the following situation: At least one neighbor of \(p\) changed its state between \(\gamma_i\) and \(\gamma_{i+1}\), and this change effectively made the guard of all actions of \(p\) false.) Similarly, an action \(A\) is said to be enabled (in \(\gamma\)) at \(p\) if the guard of \(A\) is true at \(p\) (in \(\gamma\)). We assume an unfair and distributed daemon. The unfairness means that a processor \(p\) may never be chosen by the daemon to execute an action even if it is continuously enabled unless it is the only enabled processor.

A computation step is a transition between two configurations where the transition contains at least one action and at most one action per processor. The distributed daemon implies that during a computation step, if one or more processors are enabled, then the daemon chooses at least one (possibly more) of these enabled processors to execute an action.

In order to compute the time complexity measure, we use the definition of round \([7]\). This definition captures the execution rate of the slowest processor in any computation. Given a computation \(e\), the first round of \(e\) (let us call it \(e')\) is the minimal prefix of \(e\) containing the expression of one action (an action of the protocol or the disable action) of every continuously enabled processor from the first configuration. Let \(e''\) be the suffix of \(e\), i.e., \(e = e'e''\). Then second round of \(e\) is the first round of \(e''\), and so on.

**Snap-stabilization:** We assume that in a normal execution, at least one processor (called, the initiator) initiates the protocol upon an external (w.r.t. the protocol) request by executing a special type of action, called an initialization action.

**Definition 1 (Snap-Stabilization).** Let \(P\) be a protocol designed to solve a task \(T\). \(P\) is called snap-stabilizing if and only if, starting from any configuration, any execution \(E\) of \(P\) always satisfies the specification of \(T\).
**Specification 21 (BST Problem)** A protocol $P$ is considered as a BST algorithm, if and only if the following conditions are true: (i) Any computation initiated by the root terminates in finite time. (ii) When the computation terminates, the values in the tree satisfy the BST property.

**Remark 1.** To prove that a BST algorithm is snap-stabilizing, we have to show that every execution of the protocol satisfies the following two properties: (i) starting from any configuration, the root eventually executes an initialization action. (ii) Any execution, starting from this action, satisfies Specification 21.

The time needed to reach the configuration where the initialization action is enabled is called the delay of the protocol.

### 3 Binary Search Tree Algorithm

In this section, we describe the data structures used, followed by a detailed explanation of how the algorithm works when the initiator (the root process) starts the algorithm until the values are arranged in the tree such that it becomes a BST. We divide the algorithm code in two parts: module *Heap* (Subsection 3.1) and module *Sort* (Subsection 3.2).

A node $i$ holds four constants. The constants are: the value $V_i$ that needs to be sorted in the tree, the parent ID $p_i$, the left child ID $left_i$, and the right child ID $right_i$. If $i$ does not have any of the above three neighbors, the corresponding constant's value is represented as $\bot$. For example, for the root node $r$, $p_r = \bot$, and for the leaf nodes, $left_i = right_i = \bot$. We denote the set of neighbors and set of children of $i$ by $N_i$ and $D_i$, respectively. We assume that the tree has $n$ nodes and has a height of $h$. Let $T_i$ be the subtree rooted at $i$. Then $s_i$ and $h_i$ denote the number of nodes and height, respectively, of $T_i$.

Our BST construction is transparent to the changes (addition or removal of notes) in the tree structure. If such changes occur, then the algorithm will incorporate the changes “on the fly” by nodes either entering an abnormal situation with respect to their new neighbors, or by completing the current cycle and restarting a new cycle with added/deleted values. We assume that after the add/remove operations/queries are executed, our algorithm will be initiated by the root and a new BST tree will be constructed in $O(n)$ rounds. This makes the lower bound of $\Omega(n)$ under the constraints considered in this work higher than that of the usual functions (e.g., find, insert, and delete) for a BST.

The basic idea of the algorithm is as follows: The algorithm runs in two phases. The root initiates the BST computation by starting a heappify process (shown as Module *Heap* in the algorithm) to create a maxheap of the tree. Then the root initiates the second phase (shown as *Sort* module). During this phase, the values are placed in the nodes in the BST order, placing the highest value first, the second highest value next, and so on. As the maxheap has been created in the previous phase, the root holds the maximum value of the tree. This highest value is sent to the rightmost node (say, $i$) of the tree. The destination of the
second highest value (say, second) is dependent on if \(i\) is a leaf or an internal node. If \(i\) is a leaf node, then second is sent to the parent of \(i\) (say, \(j\)). Then the third highest value (say, third) will be sent to the left child of \(j\) (if present) or to the parent of \(j\). If \(i\) is an internal node, then second goes to the left child of \(i\). Thus, values are placed in the tree following a right-parent-left order.

The algorithm will be similar if we have constructed a minheap instead of the maxheap. In that case, in the second phase, the values will be placed following a left-parent-right order. From now on, heap will imply maxheap. If a node \(i\) satisfies the maxheap property with respect to its parent and children, we say \(i\) is in heap order or in HQ in short.

Some of the variables used by a node \(i\) are described below. The rest of the variables will be defined in the informal explanations in the next two subsections. The sorted value \(SV.i\) will contain the final sorted value at the end of the algorithm. \(tSV.i\) is used to store a temporary sorted value. The heap value \(HV.i\) is the result of the first phase (Heap module). The module Sort needs to maintain the size of the subtrees rooted at each node. This size variable \(s.i\) for node \(i\) is computed in Heap and used in Sort.

A node may use at most seven states (see Figure 1 below). Module Heap uses six states: \(C\) (cleaning state), \(B\) (ready to start the heapify process), \(M\), \(M^{left}\), \(M^{right}\) (the states corresponding to if the maximum heap value \(HV\) is based on its own heap value, the maximum heap value of its left child, the maximum heap value of its right child, respectively), \(P\) (the Heap phase finished at this node, and the Sort phase is ready to start at this node). Module Sort uses \(C\), \(P\), and \(T\) (the algorithm is terminated).

![Fig. 1. The seven states used by the algorithm.](image)

A configuration in which the root is in state \(C\) is called a clean configuration. Starting from such a configuration, all other nodes in the tree will eventually reach \(C\) state. If all nodes are in \(C\) state, then the corresponding configuration is termed as a normal starting configuration. Any configuration reachable from a normal starting configuration by executing the algorithm guards is called a normal configuration. All other configurations are considered to be abnormal.

Some abnormal configurations can be locally detected by the processors. This local detection is implemented using the abnormal predicates in Algorithms 3.1 and 3.2. These predicates are used as guards of correction actions in order to avoid possible deadlocks and to speed up the protocol. Unfortunately, some problems of abnormal configurations cannot be locally detected. For example,
the initial configuration may contain some sorted values (in $tSV$) that do not match any $V$ values. The correction actions can remove the locally detectable problems in $O(h)$ rounds even before the root executes its initialization action. The other problems are eventually removed during the suffix of the protocol starting from the initialization action of the root.

Starting from an abnormal configuration, an execution not necessarily will bring the system to a normal starting configuration, but to a normal configuration. When a node has an abnormal predicate enabled, it will change its state to $C$, and all the nodes in its subtree will enter $C$ state, but not necessarily its parent (e.g. if the parent state is $B$).

Starting from a normal configuration where the root is able to execute the initialization action with no delay, the tree will become a BST in $O(n)$ rounds. In general, the worst delay is $O(n)$ rounds because the worst initial configuration is the one where no node has any of the abnormal predicates enabled, but there is a node with an incorrect $tSV$ value (that does not match any $V$ values). Thus, the abnormal configurations do not increase the asymptotic time bound. So, starting from any configuration, the tree will become a BST in $O(n)$ rounds.

The interface between the two layers (application and BST) at a node $i$ is implemented by two variables: input value to the sorting protocol $V,i$ and the final or output sorted value $SV,i$. However, every time the BST protocol runs, we do not want to disturb the application layer by writing (or overwriting) the value of $SV,i$ unless the value has changed. So, when the BST protocol terminates, $i$'s sorted value is first placed in $tSV,i$. Then $tSV,i$ and $SV,i$ are compared. The value of $tSV,i$ is copied into $SV,i$ only if the values are different (see Actions $rP3$, $iP3$, and $lP1&3$ of module $Sort$).

### 3.1 Constructing the Heap

Upon receiving an external command to sort, if the root is enabled to start the BST protocol, it starts the heapify process (module $Heap$). The root is enabled to initiate if it is in $C$ and its children are in $C$. The root broadcasts the heapify command by changing its state to $B$. As this message (wave) goes down the tree, all internal nodes change their state from $C$ to $B$. When this broadcast wave reaches the leaf nodes, they change their state from $C$ to $M$ to initiate the heapify process (or wave). During this upward wave, the nodes compute two things: the heap value (the maximum value in their subtrees) and the size of their subtrees. When this wave reaches the root, the root changes its state to $M$ and the heap is created. The root then initiates another top-down wave by changing its state from $M$ to $P$. The next phase, i.e., the BST construction phase starts from the $P$ state. We now describe the heap construction in more detail by referring Algorithm 3.1.

1. *(Start building a Heap)* If the root is in $C$, its children will change to $C$ in at most one round. Either Action $aCm$ or $aCb$ is enabled, and since it is the only enabled action, it is eventually executed in at most one round. When its children change to $C$, the root changes its state from $C$ to $B$ and sets $HV$ to its internal value $V$ (Action $CB$). An internal node changes its state from $C$ to $B$
when its parent is in $B$ and its children are in $C$. An internal node also initializes its heap value $HV$ with its input (or initial) value $V$ (Action $CB$).

Figure 2(a) shows the clean configuration for a 11-node binary tree. After $B$ wave is executed top-down, the tree state is shown in Figure 2(b). We show only the node’s internal value $V$, state $S$, and heap value $HV$. Symbol * means that the value is not important.

![Figure 2](image)

**Fig. 2.** Initial stage of constructing the heap.

2. *(Calculating heap and s.i values)* A leaf node $i$ changes its state from $C$ to $M$ and executes macro $\text{init}(i)$ (Action $CM$). In the macro $\text{init}(i)$, the node $i$ sets the size of its subtree, $s.i$ to 1 and sets the heap values of its left ($lHV$) and right ($rHV$) subtrees to $\bot$ (indicating a non-existent value).

When a parent of a leaf node detects that all its children are in state $M$ (Action $B M^*$ is enabled), it executes macro $\text{init}(i)$, change from $B$ to $M$, and executes macro $\text{set.HVs}(i)$. If the (parent) node holds a value smaller than any of the heap values of its children, it chooses as its heap value the larger heap value ($lHV$ or $rHV$) among its children and pushes its own heap value ($HV$) toward the child that was holding the larger heap value. This heapification process goes up the tree until it reaches the root.

Predicate $\text{update.HVs}(i)$ is true when due to the heapification process at the parent of $i$, i’s heap value became smaller than the values of its children. So, $HV.i$ needs to be swapped with that of one of its children. Predicate $\text{h.order}(i)$ is true if $i$ satisfies the heap property with respect to its children.

For a non-leaf node $i$ that is about to execute the macro $\text{set.HVs}(i)$, we consider three cases.

**Case 1**) $HV.i$ is larger than the heap values of its children. So, heap order is maintained at $i$. Then the macro $\text{set.HVs}(i)$ does not change the variables $S.i$ (remains $M$) and $HV.i$.

**Case 2**) Assume that the heap value of one of the children (say, the right child $\text{right}.i$) of $i$ is higher than both $HV.i$ and that of the left child of $i$. The macro $\text{set.HVs}(i)$ selects $\text{dir}.i = \text{right}$ and sets $S.i = M^{\text{right}}$. So, node $i$ will push its old heap value (now in variable $\text{down}.i$) to its right child. Assume that $\text{down}.i$ is larger than the heap values of the children of $\text{right}.i$. So, $\text{down}.i$ (the old value of $i$) needs to be pushed only one level down the tree where it becomes the new...
heap value of \( \text{right}.i \) in at most two rounds: First Action \( lrM \) is performed at \( \text{right}.i \), then Action \( M^{ir}M \) is executed at node \( i \) (\( i \) changes its state back to \( M \)). Figure 3 shows a part of a binary tree to illustrate this case. For each node we show the variable \( s \), the state \( S \), \( LHV \), \( HV \), \( rHV \), and \( \text{down} \). Symbol \( b \) means \( \perp \). The checkmark symbol marks an enabled node.

![Figure 3. Macro set\(HV\)s executed at the node with \( V = 145 \)](image)

**Case 3)** Similar to Case 2 except that \( \text{down}.i \) is smaller than the heap value of one of the children of \( \text{right}.i \). So, the old value of \( i \) (now in \( \text{down}.i \) needs to be pushed at least two levels down the tree before it finds a node \( j \) where \( \text{down}.i \) becomes the heap value of \( j \). In Figure 4, the value 130 is pushed down two levels. For each node we show the variable \( s \), the state \( S \), \( LHV \), \( HV \), \( rHV \), and \( \text{down} \). Symbol \( b \) means \( \perp \). The checkmark symbol marks an enabled node.

![Figure 4. Macro set\(HV\)s executed at node with \( V = 130 \)](image)

Smaller values may be pushed to a node \( i \) from its ancestor. When that happens, \( i \) changes its state from \( M \) to \( M^{left}/M^{right} \). When the wave (changing state from \( B \) to \( M \)) reaches the root, the root changes its state from \( B \) to \( M \). Then the root may change to state \( M^{left} \) or \( M^{right} \) if it needs to push its heap value (which is its internal value and now in \( \text{down}.r \)) down the tree. Then it pushes \( \text{down}.r \) to either \( M^{left} \) or \( M^{right} \). When the corresponding child of the
root receives the value \textit{down}, the root goes back to \textit{M} and stays in \textit{M} since it has no ancestors.

3. (Finishing the heap construction) Predicate \textit{consistency}(i) is \textit{true} when the heap values of the children of \textit{i} stored at \textit{i} are the same as the heap values stored at the corresponding children. When the root and its children are in state \textit{M} and \textit{consistency}(r) is \textit{true}, the root changes its state to \textit{P} and executes macro \textit{init}_P(r) (Action \textit{MP}). Eventually, every node changes its state from \textit{M} to \textit{P}. This \textit{P} wave eventually reaches the leaves. The root initiates the BST construction when the root and its children are in \textit{P}, i.e., the root can start the next phase even if note all nodes of the tree are in \textit{P} state.

Starting from the clean configuration presented in Figure 2(a), after executing the \textit{Heap} module when the root and its children are in state \textit{M}, a possible configuration is given in Figure 5(a). The root, when surrounded by \textit{M} state children, changes its state to \textit{P} (Figure 5(b)). For each node we show the variable \textit{s}, the state \textit{S}, \textit{IHV}, \textit{HV}, \textit{rHV}, and \textit{down}. Symbol \textit{b} means \textit{↓}.

![Diagram](image)

(a) Root and its children are done. (b) \textit{P} wave starts from the root.

Fig. 5. The root and its children are done executing Module Heap.

We defined various \textit{abnormal} predicates to characterize different types of local inconsistencies at a node during the heap construction. If any of these predicates is \textit{true} at a node, then the only enabled action at that node will be \textit{aCM}. This action when executed changes the state of the node to \textit{C}.

3.2 Constructing the BST

At the end of the heap construction, every node changes its state from \textit{M} to \textit{P} and executes the macro \textit{init}_P(i). In this macro, every non-leaf node \textit{i} sets the variable \textit{diri} to point to the child that will receive the sorted value from the root. Recall that the sorted values are placed in right-parent-left order.

Every node (including the root) will receive a sorted value from the root and send its heap value to the root. These two actions are executed concurrently. Upon completion of the heap, the root holds the maximum (heap) value of the entire tree, its children hold the maximum (heap) values of their subtrees, and so on. The above heap property is exploited in the BST construction. The root
Algorithm 3.1 Module Heap

**Predicate**

- abnormal(M) is true when the node, in state $B$, is in abnormal situations with some edgeless (parent or child).
- abnormal(M') is true when the node, in state $M'$, either has some variables with abnormal values or is in abnormal situations with some edgeless (parent or child).
- consistency(r) is true when the node has the Module property.
- update(HV,i) is true when the node needs to update its heap value since some child has a higher heap value than itself.

**(Invariants)**

- $\text{init}(i) \land \text{B at } P_r \to \text{B at } P_r$
- $\text{init}(i) \land (s.r \neq \bot) \to \text{HV}_r \neq \bot$
- $\text{init}(i) \land (s.r \neq \bot) \to (s,r) \preceq (s',r')$
- $\text{init}(i) \land (s,r \neq \bot) \land \text{ HV } = \bot \to \text{HV } = (s',r')$

**Procedure for the root node**

- $S_r$ is the data at the root in state $B$
- $s_r$ is the data at the root in state $M'$
- $H V_r$ is the heap value at the root

**Program for an internal node $i$, which is the $d$-child of its parent $i \in \{l f l, r i g h t\}$**

1. $S, j = C \land \forall j \in D, S, j = C \implies S, j = B; \text{HV}_r = V_c$
2. $M_r \leftarrow \text{abnormal}(B) \land S, j = B \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, j = M; \text{HV}_r = V_c$
3. $M' \leftarrow \text{abnormal}(M') \land S, j = B \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, j = M; \text{HV}_r = V_c$
4. $M' \leftarrow \text{abnormal}(M') \land S, j = B \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, j = M; \text{HV}_r = V_c$
5. $M' \leftarrow \text{abnormal}(M') \land S, j = B \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, j = M; \text{HV}_r = V_c$

**Program for a leaf node $i$, which is the $d$-child of its parent $i \in \{l f l, r i g h t\}$**

1. $S, i = C \land S, j = B \implies \text{HV}_r = V_c; \text{init}(i); S, i = M$
2. $M \leftarrow \text{abnormal}(M') \land S, i = M \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, i = M$
3. $M' \leftarrow \text{abnormal}(M') \land S, i = M \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, i = M$
4. $M' \leftarrow \text{abnormal}(M') \land S, i = M \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, i = M$

**Program for a leaf node $i$, which is the $d$-child of its parent $i \in \{l f l, r i g h t\}$**

1. $S, i = C \land S, j = B \implies \text{HV}_r = V_c; \text{init}(i); S, i = M$
2. $M \leftarrow \text{abnormal}(M') \land S, i = M \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, i = M$
3. $M' \leftarrow \text{abnormal}(M') \land S, i = M \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, i = M$
4. $M' \leftarrow \text{abnormal}(M') \land S, i = M \land \forall j \in D, S, j \in \{M, M'J, M' \text{ for } M' \text{ right} \} \implies \text{init}(i); S, i = M$

first sends out its own heap value to the rightmost place in the tree. The root then gets the second highest value of the tree easily (in constant steps) from one of its children. So, the concurrency of the two main tasks — sending the sorted value to the proper place and moving the heap values upward toward the root — are achieved by using the heap property. That is the reason of using the heap phase as a pre-processing phase of the BST construction.

When a sorted value sent to a node belongs to that node (i.e., it is the node's sorted value), it is stored in $t S V$. A node is done sorting if all nodes in its subtree (including itself) received their final sorted values. This is checked in the predicate done. When a node is done, it changes its state to $T$. Obviously, this wave of state change from $P$ to $T$ starts from the leaves and ends at the root. When the root changes its state to $T$, the algorithm terminates. In the following, we describe a normal execution of module Sort:

4. **(Select sorted values for all nodes)** Predicate $\text{new\_sorted}()$ is true if the root still has values to sort: $H V_r \neq \bot$, either it just started or the previous sorted value has been delivered (down $x = \bot$), there are nodes that need more sorted values ($s.r > 0$), and it has consistency with its children ($\text{consistency}(r) = \text{true}$).

If the root is in $P$ and Predicate $\text{new\_sorted}()$ is true, the only enabled action is Action $rP1$. So, it will eventually be executed. The current $H V_r$ value
is moved into $down_r$, $s.r$ is decremented, and $HV.r$ becomes $\bot$. Then the larger of the heap values of one of its children is moved in $HV.r$ by executing the macro $moveHV(r)$. That will enable Action $rP1$ again.

5. **(Receive sorted value and/or collect heap value)** Although these two actions are executed concurrently, we present them separately below:

5.1 **(Receive sorted value)** We first define a target node for some node. For some node $i$, if the condition $s.i > 0 \lor (s.i = 0 \land \text{left}.i \neq \bot \land \text{left}.i = 1)$ is true, then there exists a unique node $j$ to which $down.i \neq \bot$ will be delivered (either $j = i$ or $j$ is one of the children of $i$). We call node $j$ the current target of node $i$. $dir.i$ holds the value $j$.

We use the following predicates in this part of the algorithm:

Predicate $sent\_sorted(i)$ is true if the non-root node $i$ has previously received another value from its parent $p.i$ and it has already delivered it.

Predicate $my\_turn(i)$ is true if it is the turn of node $i$ to collect its sorted value. Node $i$ has no current sorted value ($tSV.i \neq \bot$), either it has no right subtree ($right.i = \bot$) or is full ($s.right.i = 0$), and has a value ($down.i \neq \bot$) that was not taken by any of the children of node $i$ (Predicate $sent\_sorted(i)$ is false).

Predicate $get\_sorted(j)$ is true if $j$, the $d \in \{left, right\}$ child of its parent $i$, is allowed to copy in $down.i$ the value stored at its parent, $i$ still needs sorted values ($s.j > 0$), it is the current target of node $i$ ($dir.i = d$), and the sorted value held by $i$ is a new one ($down.i \neq down.j$).

During the BST construction, if Predicate $my\_turn(i)$ is true, the target of node $i$ is $i$ itself. Otherwise, for a non-root node $i$, the target $j$ of node $i$ is one of the children of $i$ that is allowed to copy into $down.j$ the value stored at $down.i$ if either Action $iP1$ or $IP1&3$ is enabled and executed. We now consider the three types of target node $j$ of node $i$ (root, internal, and leaf) below:

[Root] The target node is the root itself ($i = j = r$). Then $my\_turn(r)$ is true and the only enabled action is Action $rP3$. The root moves $down.r$ into $tSV.r$, updates $SV.r$ if necessary, and selects its left child (if exists) as its current target (by changing $dir.r$ to left), and sets $down.r$ to $\bot$.

[Internal] The target node $j$ is an internal node. We have two cases for $j$:

1) If $my\_turn(j)$ is true, then the only enabled action for $j$ is Action $iP3$ which is similar to Action $rP3$.

2) If $my\_turn(j)$ is false, then the only possible enabled action for $j$ is Action $iP1$. $iP1$ is enabled if $get\_sorted(j)$ is true and the condition $down.j = \bot \lor sent\_sorted(j)$ are true.

Condition $down.j = \bot \lor sent\_sorted(j)$ is true if either $j$ has never received a value from $i$ ($down.j = \bot$), or has previously received another value from $i$ and has already delivered it ($sent\_sorted(j)$ is true).

When Action $iP1$ is performed, $down.i$ is copied into $down.j$, $s.j$ is decremented, and $j$ checks if it has to give up its heap value to its parent (Predicate $moveup\_HV(j)$ is explained below).
[Leaf] The target node \( j \) is a leaf node. Since the only value the leaf is allowed to receive is its own sorted value, the target of \( j \) is itself. If \( \text{get\_sorted}(i) \) is \text{true}, the only action enabled at \( j \) is \( lP1\&3 \), so it eventually gets executed.

5.2 (Collect heap value) Predicate \( \text{moveup\_hv(i)} \) is \text{true} for some node \( i \) if \( i \)'s heap value \((HV.i \neq \bot)\) was taken by \( p,i \) as its heap value. In that case, \( i \) selects the larger of the heap values of its children as its next heap value. Variable \( dhv.i \) indicates which child (heap value) will be selected. We now need to distinguish three cases.

Case 1 \([HV.i = \bot] \) \( i \) waits until \( \text{done}(i) \) becomes \text{true} so that it can change to state \( T \).

Case 2 \([HV.i \neq \bot \wedge lHV.i = \bot \wedge rHV.i = \bot] \) If node \( i \) is the root node \( r \), then it has to wait until Action \( rP1 \) becomes enabled and gets executed. Then \( HV.i \) becomes \( \bot \) and Case 1 becomes applicable.

If \( i \) is a non-root node, then when \( \text{moveup\_hv(i)} \) is \text{true}, the heap value \( HV.i \neq \bot \) is moved up the tree from node \( i \) to its parent \( p,i \) (action \( iP1, iP4, lP1\&3, \) or \( iP4 \) is enabled and executed). \( HV.i \) becomes \( \bot \) and Case 1 becomes applicable.

Case 3 \([HV.i \neq \bot \wedge (lHV.i \neq \bot \lor rHV.i \neq \bot)] \) Node \( i \) is a non-leaf node, and there exists a unique node \( j \) (decided in macro \( \text{move\_HV}s(i) \)) that will move its heap value to \( i \)'s heap value when one of Actions \( rP1, iP1, iP4, lP1\&3, \) and \( iP4 \) is executed. Node \( j \) is one of the children of \( i \) and is called the current sink of node \( i \).

If node \( i \) is the root node \( r \), then it has to wait until Action \( rP1 \) becomes enabled and gets executed. Then macro \( \text{move\_HV}s(r) \) is executed and \( HV.i \) receives the larger of the heap value of its children. Either Case 2 or Case 3 becomes applicable then.

If \( i \) is a non-root node, then if \( \text{moveup\_hv(i)} \) is \text{true}, the heap value \( HV.i \neq \bot \) is moved up the tree from node \( i \) to its parent \( p,i \) and \( i \) executes macro \( \text{move\_HV}s(i) \) (Action \( iP1 \) or \( iP4 \) is enabled and executed). Either Case 2 or Case 3 becomes enabled.

For example, starting from a configuration where the root and its children are in state \( P \), after executing the action \( rP1 \), we obtain the configuration shown in Figure 6(a). Now its right child has to execute \( iP1 \) before the root is able to move again and execute \( rP2 \). Also its left child has to execute \( iP4 \) before the root can execute \( rP1 \). Once both children execute, we obtain the configuration as shown in Figure 6(b). For each node, we show the variable \( s \), the state \( S, lHV, HV, rHV, \text{down}, \) and \( \text{dir} \). Symbol \( b \) means \( \bot \).

6. (Sets its own sorted value and adjusts the direction for the future sorted values) If \( \text{my\_turn}(i) \) is \text{true}, \( i \) collects its sorted value and adjusts the direction of sorted values toward its left subtree, if it exists. Otherwise, \( \text{dir}(i) \) is set to \( \bot \).

7. (Terminating the BST) Predicate \( \text{done}(i) \) is \text{true} when node \( i \) has \( HV.i = \bot \), does not need more sorted values from the root, and has its currently sorted value \( tSV.i \neq \bot \).

When a leaf node \( i \) is done receiving the sorted values (\( \text{done}(i) \) is \text{true}), it changes its state from \( P \) to \( T \). When a non-leaf node \( i \) is done receiving its sorted
values (predicate \textit{done}(i) is true) and all its children are in state \( T \), \( i \) changes its state from \( P \) to \( T \) (Action PT is enabled).

We defined various \textit{abnormal} predicates to characterize different types of local inconsistencies at a node during the BST construction. If any of these predicates is \textit{true} at a node, then the only enabled action at that node will be \( aCb \). This action when executed changes the state of the node to \( C \).

4 Proof of Correctness

Due to lack of space, we give a brief summary of the correctness proof, while details are available in [1].

We first present the proof of correctness assuming the weakly fair daemon. (A daemon is \textit{weakly fair} if a continuously enabled process will be eventually chosen by the daemon.) Later in Section 4.2, we show that the algorithm works under the unfair daemon as well. The time and space complexity of the algorithm are discussed in Section 4.1.

We first show a lower bound of \( \Omega(n) \) on the time complexity for the BST problem under the constraint as discussed earlier (Lemma 1).

Next, we show how the algorithm corrects any abnormal configuration into a normal configuration in finite number of rounds. Considering faulty networks, the system may start in an \textit{abnormal configuration} where there exists at least one \textit{abnormal processor}. We prove that if some node \( i \) is abnormal, then \( S,i \) becomes \( C \) in at most one round. Using this result, we show that if \( S,i = C \), all the nodes in the subtree rooted at \( i \), \( T \) change to \( C \) in \( O(h_i) \) rounds. Then in \( O(h) \) rounds the system reaches a configuration which does not contain any local problem and the behavior of the protocol is now almost as the normal behavior (the result is in Lemma 4. We conclude that the delay (the time needed for the root to execute the initialization action) of our algorithm is \( O(n) \) rounds (Lemma 2).

Once we establish the finite round delay (as above), our remaining obligation is to show that starting from a normal starting configuration, the tree will satisfy the BST property in finite rounds.
We prove that starting from a normal starting configuration, the state of every node eventually becomes $B$ (for non-leaf nodes) or $M$ (for leaf nodes). Any internal node sets its heap value and changes its state from $B$ to $M/MLeft/MRight$. When the root $r$ is in $M$ state, then $HVR$, $IHVR$, and $rHVR$ hold the maximum value in the entire tree, in its left subtree, and in its right subtree, respectively. We conclude that, starting from a normal starting configuration, in at most
4h + 3 rounds, the tree will satisfy the heap property and all nodes will be in state $P$ (Lemma 3).

Note that the guarded actions of Module Sort for each process (root, internal, and leaf node) are mutually exclusive. So, at any time during the BST construction, at most one of the root actions is enabled, at most one of any internal node actions is enabled, and at most one of any leaf node actions is enabled. This was done to implement a sequential, pipelined delivery of sorted values in decreasing order.

The root will continue sending the sorted values (via $\text{down}_r$) in descending order as long as there exists a target node for a value ($\text{new\_sorted}()$ will be true as long as $s.r > 0$). The value of $\text{down}_r$ follows a path of current target nodes, starting from the root and ending at some node in the tree. When the root takes the heap value of its sink node to make it a sorted value, in at most one round, the child adjust its heap value to one of its children. In at most $n$ rounds, the root is done generating sorted values, and in at most additional $h$ rounds every node in the system receives its sorted value (Lemma 4), and enters state $T$. Once the root enters $T$ state, the BST construction is done.

Finally, we prove that starting from an arbitrary configuration where the nodes have distinct internal values drawn from an arbitrary set, our algorithm arranges them in a BST order in $O(n)$ rounds (Theorem 1).

Theorem 1 and Lemma 1 imply that the proposed BST algorithm is time optimal.

**Lemma 1.** Under the space constraint that the maximum number of items that can be stored at any time at any processor is constant (i.e., independent of $n$), the lower bound on the time complexity for arranging $n$ values in a given tree in a distributed manner such that the tree becomes a binary search tree (BST) is $\Omega(n)$.

**Proof.** Assume that all the values larger (respectively, smaller) than the root’s value are currently in the left (respectively, right) subtree of the root. Then $n - 1$ values have to pass by the root to move to their right place in the BST. As the root has a constant memory, it will require the root to execute at least $n$ actions to move those values.

**Lemma 2.** The delay is in $O(n)$ rounds.

**Lemma 3.** Starting from a normal starting configuration, it takes at most $4h + 3$ rounds to heapify the tree.

**Lemma 4.** Starting from a normal configuration, where the root and its children are in state $P$, in at most $n + h$ rounds, every node in the system receives its sorted value and changes its state to $T$.

**Theorem 1.** Starting from an arbitrary configuration where $n$ values are arranged in a binary tree, each node holding a single key value, Algorithm 3.2 arranges those $n$ values such that the tree becomes a BST in $O(n)$ rounds and requires $O(\log n)$ space.
4.1 Complexity

All the variables used by a node \( i \) except one \((s,i)\) require \( O(1) \) space complexity. Variable \((s,i)\) requires \( O(\log s_i) \) bits where \( s_i \) represents the total number of nodes in the subtree \( T_i \) rooted at \( i \). In the worst case, (for the root node) the space complexity is \( O(\log(n)) \). The heap construction does not use this variable, it is needed only for the BST construction. But, just for better presentation, we included the the computation of \( s,i \) in Module Heap. Therefore, we claim that the heap construction uses \( O(1) \) space. The BST construction requires \( O(\log n) \) space in the worst case.

Both the time complexity and delay of the proposed algorithm is \( O(n) \) rounds.

Starting from a normal starting configuration, the time to arrange the values as a heap is \( O(h) \) (at most \( 4h + 3 \) rounds)(Lemma 3).

Starting from an arbitrary configuration, the time to arrange the values such that the tree is a BST order is \( O(n) \) rounds (Theorem 1).

4.2 Unfair Daemon

In any round, the total number of actions executed by all processes is bounded. Since any execution of our algorithm has a bounded complexity in terms of steps (or actions), the total number of actions executed in a normal execution is bounded. Thus, the duration of a round cannot be extended forever by ignoring some enabled processes for an indefinite period of time.

5 Conclusion

We present the first snap-stabilizing distributed binary search tree (BST) algorithm and the first snap-stabilizing heap algorithm. A key feature of both the solutions is that the maximum number of items that can be stored at any time at any processor is constant — independent of the size \( (n) \) of the network.

The proposed snap-stabilizing distributed BST solution ensures that when a BST computation initiated by the root terminates, the tree is a BST. Under the space constraint, our BST solution is asymptotically optimal in time and takes \( O(n) \) rounds. A processor \( i \) requires \( O(\log s_i) \) bits of space where \( s_i \) is the size of the subtree rooted at \( i \). So, the root uses \( O(\log n) \) bits.

The heap construction spends \( O(h) \) rounds where \( h \) is the height of the tree. Its space requirement is constant, independent of \( n \).

If the space complexity of the heap solution is asymptotically optimal, it is an open problem to show that it is possible to design a BST protocol with a space complexity of \( O(1) \) while keeping a time complexity of \( O(n) \).

Our BST construction is transparent to changes in the tree structure (by adding/removing nodes). If such changes occur, then the algorithm will incorporate the changes “on the fly” by nodes either entering an abnormal situation with their new neighbors, or by completing the current cycle and restarting a new cycle with added/deleted values. Since the performance of the usual functions on the tree (as find, insert, delete) is directly related to the structure of the
tree (height, balanced or unbalanced property etc.) and as we have mentioned before, our algorithm does not alter the tree structure, we do not address this issue of performance here.

Also, the usual operations on the tree make sense if the tree has the BST property. Since the BST property is guaranteed when the algorithm terminates, and for stabilization purposes the algorithm is an infinite cycles of complete BST construction, these functions insert, find, delete can be applied to the output values.

References