Binary Search Trees

Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst
Sorted Sets

Data structure for a collection of items each having a \textit{distinct} key and belonging to one of several sets

- sets are identified by one of their items; initially, each item belongs to a singleton set

Operations

- \texttt{setkey}(i,k): initialize key of item \(i\) to \(k\); \(i\) must be a singleton
- \texttt{access}(k,s): return the item in set \(s\) having key \(k\)
- \texttt{insert}(i,s): insert item \(i\) into \(s\); \(i\) must be a singleton
- \texttt{delete}(i,s): remove item \(i\) from \(s\); \(i\) becomes a singleton
- \texttt{join}(s_1,i,s_2) return set formed by combining \(s_1\), \(i\) and \(s_2\); all items in \(s_1\) must have keys \(<\text{key}(i)\) and all items in \(s_2\) must have keys \(>\text{key}(i)\); operation destroys \(s_1\) and \(s_2\)
- \texttt{split}(i,s); split set \(s\) containing \(i\) into three sets: \(s_1\) containing items with keys \(<\text{key}(i)\), \(\{i\}\) and \(s_2\) containing items with keys \(>\text{key}(i)\); return pair \([s_1,s_2]\)
Sorted Sets and Binary Search Trees

- **Symmetric ordering** - items arranged in trees so that for every node $v$
  - keys of nodes in $v$'s left subtree are smaller than $key(v)$
  - keys of nodes in $v$'s right subtree are larger than $key(v)$

- To insert new key value, search for key and insert at place where search "falls out" of tree
item function access(keytype k, sorted set s);
  do s ≠ null and  k<key(s) ⇒ s:=left(s)
  | s ≠ null and  k>key(s) ⇒ s:=right(s)
  od;
  return s
end;

procedure insert(item i, sorted set s);
  item x;  x:=s;
  if s = null ⇒ return i fi;
  do key(i)<key(x) and  left(x) ≠ null ⇒ x:=left(x)
  | key(i)>key(x) and  right(x) ≠ null ⇒ x:=right(x)
  od;
  if key(i)=key(x) ⇒ return null;
  | key(i)<key(x) ⇒ left(x):=i;
  | key(i)>key(x) ⇒ right(x):=i;
  fi;
  p(i) := x;
end;
procedure delete(item i, sorted set s);
    item j;
    if left(i) ≠ null and right(i) ≠ null ⇒
        j := left(i);
        do right(j) = null ⇒ j := right(j) od;
        swapplaces(i,j);
    fi;
    if left(i) = null ⇒ left(i) ↔ right(i) fi;
    p(left(i)) := p(i);
    if i = left(p(i)) ⇒ left(p(i)) := left(i) |
    i = right(p(i)) ⇒ right(p(i)) := left(i) fi;
    left(i),right(i),p(i) := null;
    return s
end;
sset function join(sset $s_1$, item $i$, sorted set $s_2$);
    left($i$) := $s_1$; right($i$) := $s_2$;
    $p(s_1)$, $p(s_2)$ := $i$;
    return $i$;
end;

[sset, sset] function split(item $i$, sorted set $s$);
    sset $x$, $s_1$, $s_2$;
    $x := p(i)$; $s_1$, $s_2 :=$ left($i$), right($i$);
    leftchild := ($i = left(x)$)
    do $x 
eq$ null ⇒
        if leftchild ⇒ $s_2 :=$ join($s_2$, $x$, right($x$))
        | not leftchild ⇒ $s_1 :=$ join(left($x$), $x$, $s_1$)
        fi;
        leftchild := ($x = left(p(x))$); $x := p(x)$
    od;
    left($i$), right($i$), $p(i)$ := null;
    $p(s_1)$, $p(s_2)$ := null;
    return ($s_1$, $s_2$);
end;
Analysis of Binary Search Trees

- **Access** takes time proportional to depth of item
- **Insert** takes time proportional to depth of item after insertion
- **Delete** takes time proportional to depth of deleted item if it has a **null** child, and time proportional to depth of its symmetric order predecessor if it has no **null** child
- **Join** take constant time
- **Split** takes time proportional to depth of item on which the split is taking place
- Depth of BST on $n$ nodes can be $n-1$ in worst case, so most operations have worst-case running time $\Omega(n)$
  - can improve to $O(\log n)$ by balancing subtrees
Balanced Binary Trees

- A balanced binary tree (BBT) is a full binary tree with each node \( x \) having integer \( \text{rank}(x) \) that satisfies following:
  - if \( x \) is a node with a parent, \( \text{rank}(x) \leq \text{rank}(p(x)) \leq \text{rank}(x) + 1 \)
  - if \( x \) is a node with a grandparent, \( \text{rank}(x) < \text{rank}(p(p(x))) \)
  - if \( x \) is an external node, \( \text{rank}(x) = 0 \); if \( x \) also has a parent, \( \text{rank}(p(x)) = 1 \)

- Also called red-black trees
  - red nodes have same rank as parents, black nodes have different ranks
  - sufficient to store 1 bit of balance information
Depth of Balanced Binary Trees

- **Lemma 4.1.** A node of rank $k$ in a balanced binary tree has height at most $2k$ and at least $2^{k+1} - 1$ descendants; therefore, a balanced binary tree with $n$ internal nodes has depth at most $2\lg(n+1)$

  **Proof.** The proof of first part is by induction on $k$

  » basis ($k=0$) is obvious since by definition of ranks, any node of rank 0, must be external, hence its height is 0 and it has 1 descendant

  » assume lemma is true for rank $k$, and let $x$ have rank $k+1$

    • by definition of ranks and induction hypothesis, the grandchildren of $x$ have height at most $2k$, so $x$ can have height at most $2(k+1)$

    • similarly, its two subtrees must contain at least $2^{k+1} - 1$ nodes, so $x$ has a total of at least $2(2^{k+1} - 1) + 1 = 2^{k+2} - 1$ descendants

  by the first part of the lemma, the rank of the root is at most $\lg(n+1)$ and the height of the root is at most twice its rank

- **Lemma implies access time in a BBT is $O(\log n)$**
Rotation Operations

**single rotation**

```
       X
      / \
     Y   Z
    / \  / \  \\
   A   B C   D
```

**rotate(y)**

**rotate taller subtree up to reduce height discrepancy**

**double rotations**

**outer**

```
       X
      / \
     Y   Z
    / \  / \  \\
   A   B C   D
```

**rotate^2(x)**

**inner**

```
       X
      / \
     Y   Z
    / \  / \  \\
   A   B C   D
```

**rotate^2(z)**
Convenience Notations/Operations

- Special notations
  - \( p^2(x) \) – \( p(p(x)) \)
  - \( sib(x) \) – sibling of \( x \)
  - \( uncle(x) \) – child of grandparent that is not parent
  - \( nephew(x) \) – far child of sibling
  - \( niece(x) \) – near child of sibling
  - \( outer(x) \) – true if \( x \) is leftmost or rightmost grandchild
    - \( inner(x) \) – not \( outer(x) \)

- Operations
  - \( rotate(x) \) – rotates \( x \) up to parent’s position
  - \( rotate^2(x) \) – rotates \( x \) up to grandparent’s position
    - equivalent to two rotations
Insertion in BBTs
Rebalancing After Insert/Delete

In case of insertion, let \( x \) be inserted node

\[
\text{do } p^2(x) \neq \text{null and } \text{rank}(x) = \text{rank}(p^2(x)) \Rightarrow \\
\quad \text{if } \text{rank}(p(x)) = \text{rank}(\text{uncle}(x)) \Rightarrow \\
\qquad x := p^2(x); \text{rank}(x) := \text{rank}(x) + 1; \\
\qquad \text{if } \text{outer}(x) = x := \text{rotate}(p(x)); \text{done} \\
\qquad \text{inner}(x) = x := \text{rotate}^2(x); \text{done} \\
\quad \text{fi}; \\
\text{od};
\]
In case of deletion, let $x$ be root of the subtree that moved up (if any)

```plaintext
do \ p(x) \neq \text{null and } rank(p(x))=\text{rank}(x)+2 \Rightarrow \\
\quad r := \text{rank}(x) \\
\quad \text{if } rank(sib(x)) = r+1 \Rightarrow \\
\quad \quad \text{if } rank(nephew(x)) = rank(niece(x)) = r \Rightarrow \\
\quad \quad \quad x := p(x); rank(x) := r+1; \\
\quad \quad \quad \text{rotate}(sib(x)); \ rank(p(x)) = r+1; \ rank(p^2(x))=r+2; \ done \\
\quad \quad \text{if } rank(niece(x)) = r+1 > rank(nephew(x)) \Rightarrow \\
\quad \quad \quad \text{rotate}^2(niece(x)); \ rank(p(x))=r+1; \ rank(p^2(x))=r+2; \ done \\
\quad \text{fi;} \\
\quad \text{fi;}
\od;
```

![Diagram showing the deletion process](image)
Join and Split

- **join**(s₁, i, s₂): when rank(s₁)<rank(s₂)
  - find leftmost node x in s₂ with rank(x)=rank(s₁)
  - replace x with i, make s₁ the left child and x the right child of i
  - make rank(i)=rank(s₁)+1 then rebalance as in insert

- **split**(i, s) is done in same way as for unbalanced trees but using new join
  - differences between ranks of joined subtrees sums to $\leq$ rank(root), so $O(\log n)$ time

- Can use to implement split/join lists
  - $O(\log n)$ time per operation
Dual Key Search Trees

- Data structure for sets of items having two keys $key_1$ and $key_2$; key values need not be unique
  
  - `setkey(i, k_1, k_2)`: initialize keys of singleton item $i$ to $k_1, k_2$
  - `insert(i, s)`: insert $i$ into $s$
  - `delete(i, s)`: remove item $i$ from $s$; $i$ becomes a singleton
  - `access(k_1, s)`: return some item in set $s$ having $key_1 = k_1$
  - `findmin(k_1, s)`: return item in $s$ with smallest $key_2$ value among those with $key_1 \leq k_1$

- Combines aspects of search trees and heaps

- Can implement all ops in $O(\log n)$ time

- Extend to find min $key_2$ item in $key_1$ range $[lo, hi]$
  
  - can use to find min $key_2$ item in a time interval
Implementation

- Nodes arranged in binary search tree order on $key_1$
- Field $min_2$ is node in subtree with smallest $key_2$ value
- After insertion/deletion, update $min_2$ fields going up tree
- Also, update during rotation operations
Implementing `findmin`

let $key_2(null)$ be larger than any valid key, let $min_2(null) = null$

item function `findmin(keytyp k, sset2 s)`:

```plaintext
item u, v; u := s; v := null;
  do u ≠ null ⇒
      if $key_2(u) > k$, ⇒
        u := left(u);
      else if $key_2(u) ≤ k$ ⇒
        if $key_2(u) < key_2(v)$ ⇒ v := u; fi;
        if $key_2(min_2(left(u))) < key_2(v)$ ⇒ v := min_2(left(u)); fi;
      u := right(u);
    fi;
  od;
  return v;
end;
```

$v$ is best node found so far

`key_2` values of $u$ and $right(u)$ are too large, so "target" must be in $left(u)$

all nodes in $left(u)$ are "eligible" so no need to search $left(u)$

`right(u)` might contain an eligible node

`right(u)` might contain an eligible node
Exercises

1. Show the result of performing a split operation at node e on the (unbalanced) binary search tree shown below.

The split produces the three trees shown below.
2. The diagram below shows a right rotation operation at node \( x \). Show that if the *height* of subtree A is larger than the heights of subtrees B and C, the rotation reduces the height of the overall tree.

Now suppose that in the previous diagram, the height of B is larger than the heights of subtrees A and C. The diagram at left below shows the same tree but with the subtree B shown in more detail (with root \( z \) and its subtrees \( B_1 \) and \( B_2 \)). The right hand diagram is obtained from the left by a double rotation. Show that the right-hand tree has smaller height than the left.

The height of a tree is the max distance from its root to one of its leaves, so in the left hand tree, the overall height is \( 2 + \text{height}(A) \), if the height of A is larger than the heights of B and C. The height of the right hand tree, is \( \max(1 + \text{height}(A), 2 + \text{height}(B), 2 + \text{height}(C)) \) but since A has height greater than the other two, this is \( 1 + \text{height}(A) \).

The height of the left-hand tree is \( 3 + \max(\text{height}(B_1), \text{height}(B_2)) \) and the height on the right is \( 2 + \max(\text{height}(B_1), \text{height}(B_2)) \).
3. Draw a picture of a balanced binary tree on 10 nodes that is as unbalanced as it can possibly be and show a sequence of insert and delete operations that will produce that tree.

The diagram below has 11 nodes. If the nodes are inserted in alphabetical order, we get the balanced binary tree shown below. If we then remove node h, there is no further change in the tree structure and we get a maximally unbalanced 10 node tree.

4. Consider a rotation operation at y in the dual key search tree shown below. How would you update the min_2 values for nodes x and y shown below?

With the assignment

\[ \text{min}_2(x) := \min(\text{key}_2(x), \text{min}_2(B), \text{min}_2(C)) \]

followed by

\[ \text{min}_2(y) := \min(\text{key}_2(y), \text{min}_2(A), \text{min}_2(x)) \]
5. Write a program for the dual key search tree that implements an operation findminRight(k1, s) that returns the item with the smallest key2 value in s from among those items with key1 values that are ≥ k1.

```c
item function findminRight(keytyp k1, sset2 s);
  item u, v; u := s; v := null;
  do u ≠ null ⇒
    if key1(u) < k1 ⇒
      u := right(u);
    else if key1(u) ≥ k1 ⇒
      if key2(u) < key2(v) ⇒ v := u; fi;
      if key2(min2(right(u))) < key2(v) ⇒
        v := min2(right(u)); fi;
    fi;
  od;
  return v;
end;
```

How would you extend the data structure so that it would also support a findmax operation? Add a max2 field to each node that identifies the node in the subtree of the given node that has the largest key2 value. These fields can be updated in the same way as the min2 fields.