Lecture 9: March 8, 2001

Today:

- Hash deletion in $O(1)$ time
- Binary Search Trees
- BST Search
- BST Insert
- BST Delete

```c
node *insert (T, char *s) {
  int h = hash(s);
  node *ph = malloc(sizeof(char*) + 2 * sizeof(void*));
  ph.s = s;
  ph.prev = NIL;
  ph.next = T[h].head;
  if (T[h].head) T[h].head.prev = ph;
  T[h].head = ph;
  return ph;
}

node *search (T, char *s) {
  int h = hash(s);
  node *ph = T[h].head;
  while (ph)
    if (!strcmp(s, ph.s)) return ph;
    else ph = ph.next;
  return NULL;
}

void SearchAndDelete (T, char *s) {
  node *ph = search (T, s);
  if (ph) {
    ph.next.prev = ph.prev;
    ph.prev.next = ph.next;
  }
}

void delete (T, node *ph) {
  ph.next.prev = ph.prev;
  ph.prev.next = ph.next;
}
```

Binary Search Trees

- Each element $x$ in binary search tree contains:
  - $key[x]$ - key stored at $x$.
  - $left[x]$ - pointer to left child of $x$.
  - $right[x]$ - pointer to right child of $x$.
  - $p[x]$ - pointer to parent of $x$.

BST Element
Binary-Search-Tree Property

Keys in binary search tree satisfy the binary-search-tree property:

- Let \( x \) be a node in a binary search tree.
- \( y \) in left subtree of \( x \) \( \Rightarrow \) \( \text{key}[y] \leq \text{key}[x] \)
- \( y \) in right subtree of \( x \) \( \Rightarrow \) \( \text{key}[x] \leq \text{key}[y] \)

---

Inorder Tree Walk

- Can print keys in BST with inorder tree walk.
- Key of each node printed between the keys in left subtree and those in right subtree.

\[
\text{INORDER-TREE-WALK}(x) \\
\begin{array}{l}
1 \text{ if } x \neq \text{NIL} \\
2 \text{ then INORDER-TREE-WALK(left}[x]) \\
3 \text{ print key}[x] \\
4 \text{ INORDER-TREE-WALK(right}[x])
\end{array}
\]

- Prints elements in monotonically increasing order.
- Running Time: \( \Theta(n) \)

---

Inorder Tree Walk

- Inorder tree walk can be thought of as a projection of BST onto a line.
- \( 2^d \) leaves at depth \( d \) occur at \( x \) values \( \frac{1}{2^{d+1}}, \frac{3}{2^{d+1}}, \ldots, \frac{2^d - 1}{2^{d+1}} \).
Other Tree Walks

- A **preorder tree walk** processes each node before processing its children.

  **Preorder-Tree-Walk**(\(x\))
  
  1 if \(x \neq \text{NIL}\)
  2 then print \(key[x]\)
  3 \(\text{Preorder-Tree-Walk}(\text{left}[x])\)
  4 \(\text{Preorder-Tree-Walk}(\text{right}[x])\)

- A **postorder tree walk** processes each node after processing its children.

  **Postorder-Tree-Walk**(\(x\))
  
  1 if \(x \neq \text{NIL}\)
  2 then \(\text{Postorder-Tree-Walk}(\text{left}[x])\)
  3 \(\text{Postorder-Tree-Walk}(\text{right}[x])\)
  4 print \(key[x]\)

Searching in a BST

- To find element with key \(k\) in tree \(T\):
- Compare \(k\) with \(key[\text{root}[T]]\)
- If \(k < \text{key}[\text{root}[T]]\) search for \(k\) in left subtree
- Otherwise, search for \(k\) in right subtree

Search Code

Recursive:

\(\text{SEARCH}(T, k)\)

1 \(x \leftarrow \text{root}[T]\)
2 if \(x = \text{NIL}\) return NIL
3 if \(k = \text{key}[x]\) return \(x\)
4 if \(k < \text{key}[x]\) then return \(\text{SEARCH}(T, \text{left}[x])\)
5 else return \(\text{SEARCH}(T, \text{right}[x])\)

Iterative:

\(\text{SEARCH}(T, k)\)

1 \(x \leftarrow \text{root}[T]\)
2 while \(x \neq \text{NIL}\) and \(k \neq \text{key}[x]\)
3 do if \(k < \text{key}[x]\)
4 then \(x \leftarrow \text{left}[x]\)
5 else \(x \leftarrow \text{right}[x]\)
6 return \(x\)
Search Examples

- **SEARCH(T, 11)**

- **SEARCH(T, 4)**

---

**Proof of Correctness**

- $S(x) = \{\text{elements in subtree rooted at } x\}$

- **Loop Invariant:**
  
  $I = \{k \in S(root[T]) \Rightarrow k \in S(x)\}$

- Initially, $I$ is true.

---

**Proof of Correctness**

- **Inductive Step:**
  
  - $k \in S(root[T]) \Rightarrow k \in S(x)$
  
  - If $k < key[x]$, then BST property implies $k \notin S(right[x])$.
  
  - Therefore, $I \Rightarrow k \in S(left[x])$ which implies $k \in S(root[T]) \Rightarrow k \in S(x')$, where $x' = left[x]$.
**Proof of Correctness**

- Termination occurs since $|S(x)|$ strictly diminishes.
- When loop terminates, either $x = \text{NIL}$ or $k = \text{key}[x]$.
- Therefore, if $x$ in tree, SEARCH will find it.
- Otherwise, SEARCH returns NIL.

**Analysis of Search**

- Running time is $O(h)$ on tree of height $h$.
- Worst case running time is $\Theta(n)$.

**BST Insertion**

- Similar to SEARCH
- INSERT takes an element $z$ whose right and left children are NIL and inserts it into $T$.
- Find place in $T$ where $z$ belongs, using code similar to that of SEARCH.
- Add $z$ there.

**Insert Code**

Insert Code

\[
\begin{align*}
\text{INSERT}(T, z) & \quad 1 \ y \leftarrow \text{NIL} \\
& \quad 2 \ x \leftarrow \text{root}[T] \\
& \quad 3 \ \text{while} \ x \neq \text{NIL} \\
& \quad \quad 4 \ \text{do} \ y \leftarrow x \\
& \quad \quad \quad 5 \ \text{if} \ \text{key}[z] < \text{key}[x] \\
& \quad \quad \quad \quad 6 \ \text{then} \ x \leftarrow \text{left}[x] \\
& \quad \quad \quad \quad 7 \ \text{else} \ x \leftarrow \text{right}[x] \\
& \quad \quad 8 \ p[z] \leftarrow y \\
& \quad 9 \ \text{if} \ y = \text{NIL} \\
& \quad \quad 10 \ \text{then} \ \text{root}[T] \leftarrow z \\
& \quad \quad 11 \ \text{else if} \ \text{key}[z] < \text{key}[y] \\
& \quad \quad \quad 12 \ \text{then} \ \text{left}[y] \leftarrow z \\
& \quad \quad \quad 13 \ \text{else} \ \text{right}[y] \leftarrow z 
\end{align*}
\]
**BST Insertion Example**

- Insert element with key 8 into tree.

![BST Insertion Example Diagram]

**BST Sorting Algorithm**

- Can use `INSERT` and `INORDER-TREE-WALK` to sort list of \( n \) elements, \( A \).

```plaintext
BST-Sort
1 \( root[T] \leftarrow \text{NIL} \)
2 for \( i \leftarrow 1 \) to \( n \)
3 \hspace{1em} do \text{INSERT}(T, A[i])
4 \text{INORDER-TREE-WALK}(root[T])
```

**Sorting Example**

Sort the following numbers:

\[ 5 \ 10 \ 7 \ 1 \ 3 \ 1 \ 8 \]

- Build binary search tree.

![Sorting Example Diagram]

**Sorting Example Cont.**

- \text{INORDER-TREE-WALK}

\[ 1 \ 1 \ 3 \ 5 \ 7 \ 8 \ 10 \]
Sorting Analysis

- Compare with QUICKSORT partitioning around first (or \(n^{th}\) element)
- Same comparisons! (Different order).
- Therefore, \(O(n \log n)\) expected time on random input permutation.
- Average tree height is \(O(\log n)\) (recitation)

Example: 3 1 8 2 6 7 5

Minimum

- **Problem:** Find minimum key in tree rooted at \(x\).
- **Solution:** Follow left branches.

\[
\text{MINIMUM}(x) \\
1 \text{while } \text{left}[x] \neq \text{NIL} \\
2 \quad \text{do } x \leftarrow \text{left}[x] \\
3 \text{return } x
\]

- Running Time = \(O(h)\).

![Minimum Diagram]

Successor

- **Problem:** Given \(x\), find node with smallest key greater than \(\text{key}[x]\).
- Two cases.

Successor Case 1

- Right subtree of \(x\) is nonempty.
- Successor is leftmost node in right subtree.
- \(\text{MINIMUM(\text{right}[x])}\).
**Successor Case 2**

- Right subtree of \( x \) is empty.
- Successor is *lowest* ancestor of \( x \) whose left child is also an ancestor of \( x \).

![Successor Diagram]

- Careful! “Successor” is defined as the element next encountered by preorder traversal!

**Successor Pseudocode**

\[
\text{SUCCESSOR}(x) \\
1 \text{ if } \text{right}[x] \neq \text{NIL} \\
2 \quad \text{then return TREE-MINIMUM(right\[x\])} \\
3 \quad y \leftarrow p[x] \\
4 \quad \text{while } y \neq \text{NIL} \text{ and } x = \text{right}[y] \\
5 \quad \text{do } x \leftarrow y \\
6 \quad \quad y \leftarrow p[y] \\
7 \quad \text{return } y
\]

- Time = \( O(h) \).

**Deletion**

- **Problem:** Delete node \( x \) from BST.
- 3 cases
  - \( x \) has no children.
  - \( x \) has one child.
  - \( x \) has two children.

**Case 1**

- If \( x \) has no children, just remove \( x \).
Case 2

- If $x$ has exactly one child, then make $p[x]$ point to that child.

```
• Problem: worst case execution time for dynamic set operations on BST is $\Theta(n)$. No better than linked list!
• Solution: “Balanced” search trees—guarantee small height.
• Next time!
```

Delete Pseudocode

```
DELETE(T, z)
1 if left[z] = NIL or right[z] = NIL
2 then y ← z
3 else y ← TREE-SUCCESSOR(z)
4 if left[y] ≠ NIL
5 then x ← left[y]
6 else x ← right[y]
7 if x ≠ NIL
8 then p[x] ← p[y]
9 if p[y] = NIL
10 then root[T] ← x
11 else if y = left[p[y]]
12 then left[p[y]] ← x
13 else right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
16 ▷ If y has other fields, copy them, too.
17 return y
```