Dictionary data structures

William Hendrix

Lecture 10

Outline

- Review
- Other dictionary implementations
  - Binary search trees
  - Hash tables
  - Bit vectors
- Heaps
- Union-find
- Prefix and suffix trees

Dictionary

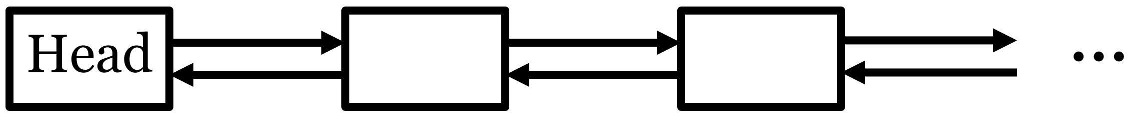
- Abstract data structure for storing and retrieving values

- **Primary operations**
  - *Search(x)*: returns the location of *x* in the dictionary, or NIL if not contained
  - *Insert(x)*: adds *x* to the dictionary
  - *Delete(x)*: removes *x* from the dictionary

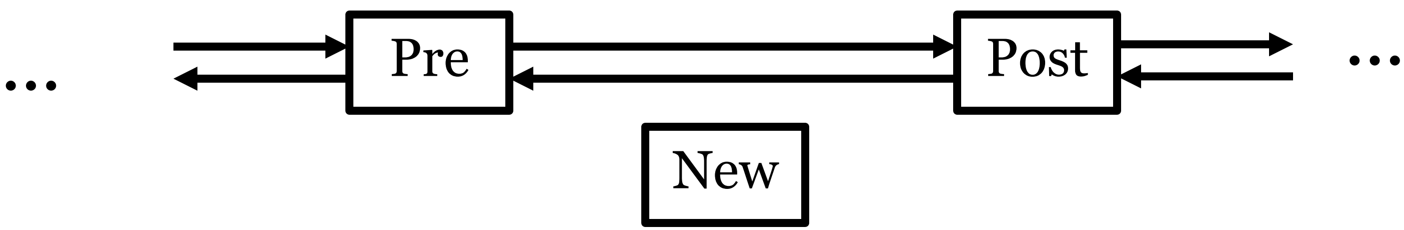
- **Additional operations**
  - *Max(), Min()*: return the location of the largest/smallest element
  - *Successor(x), Predecessor(x)*: return the next largest/smallest element than *x*

Linked list operations

- Search: linear scan

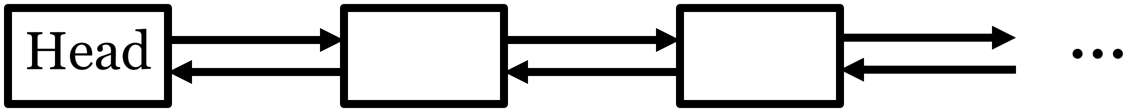


  - If sorted, stop when values are too large
  - If DLL, you can search backwards from end or forwards
- Insert: add links to include new node in chain

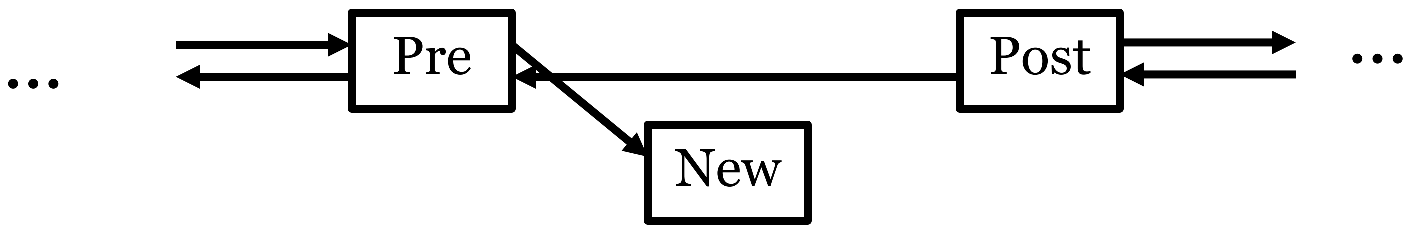


Linked list operations

- Search: linear scan

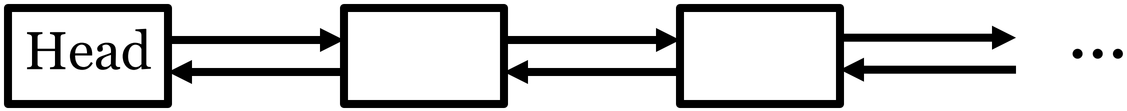


  - If sorted, stop when values are too large
  - If DLL, you can search backwards from end or forwards
- Insert: add links to include new node in chain

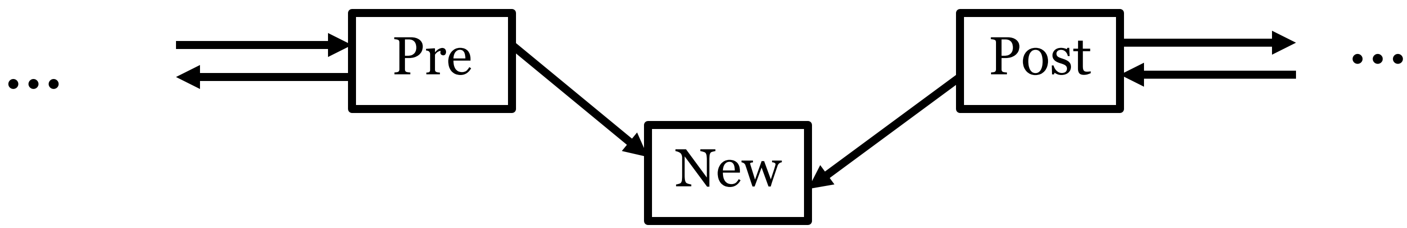


Linked list operations

- Search: linear scan

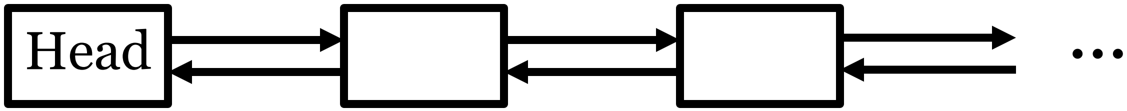


  - If sorted, stop when values are too large
  - If DLL, you can search backwards from end or forwards
- Insert: add links to include new node in chain

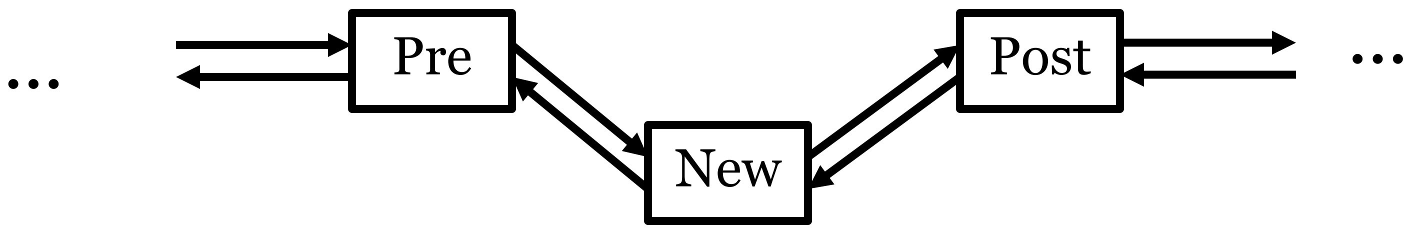


Linked list operations

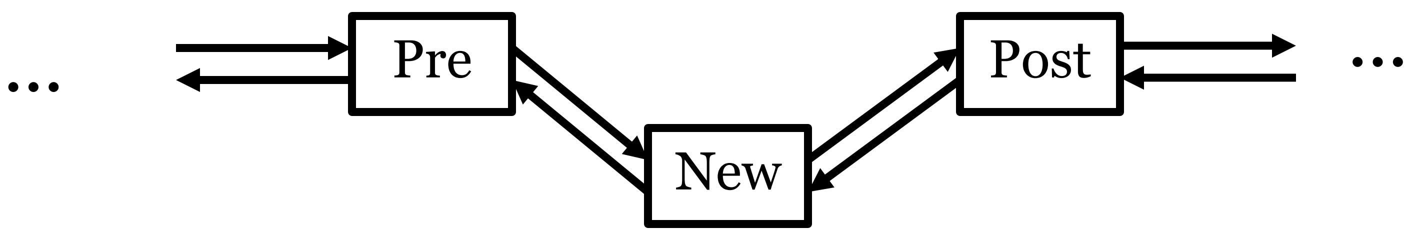
- Search: linear scan



  - If sorted, stop when values are too large
  - If DLL, you can search backwards from end or forwards
- Insert: add links to include new node in chain

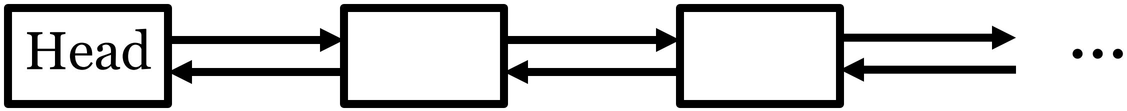


  - If sorted, scan to find insertion position
- Delete: reroute links, then delete victim

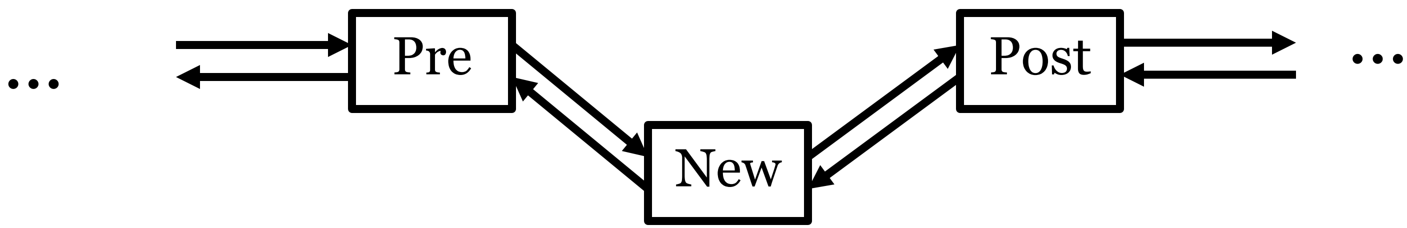


Linked list operations

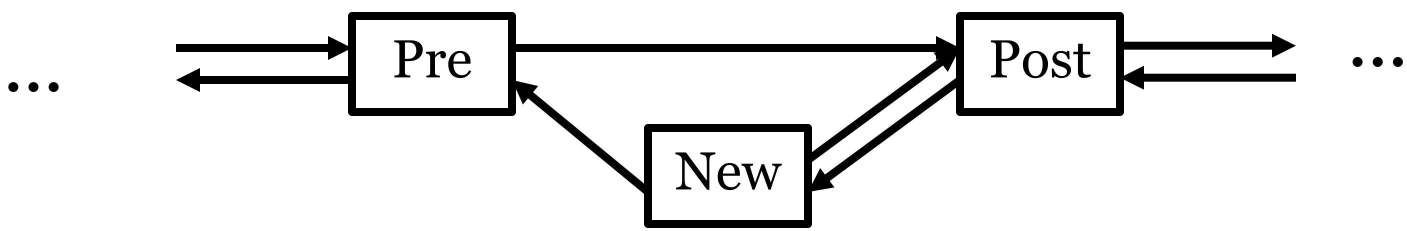
- Search: linear scan



  - If sorted, stop when values are too large
  - If DLL, you can search backwards from end or forwards
- Insert: add links to include new node in chain

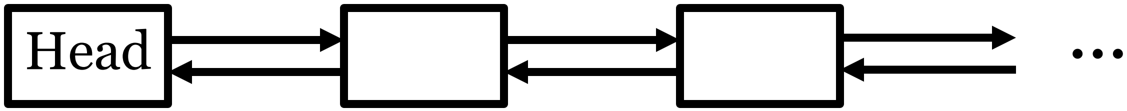


  - If sorted, scan to find insertion position
- Delete: reroute links, then delete victim

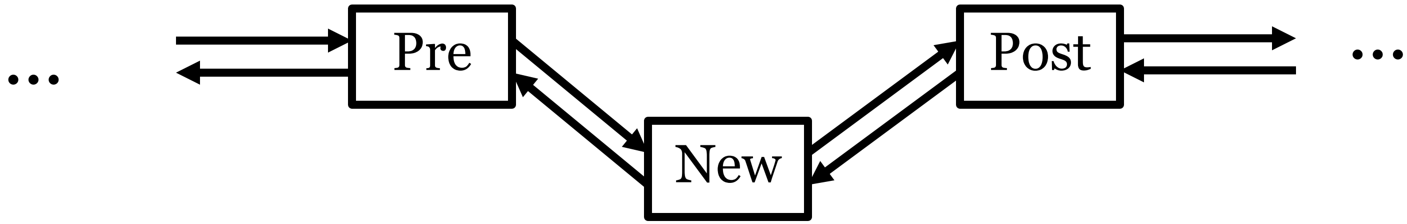


Linked list operations

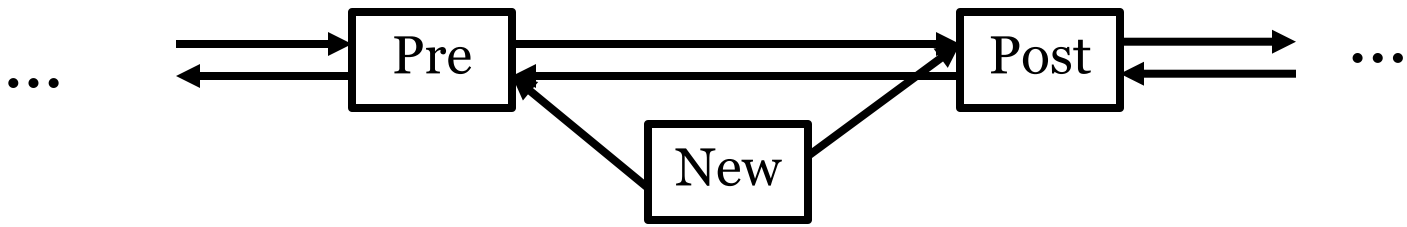
- Search: linear scan



  - If sorted, stop when values are too large
  - If DLL, you can search backwards from end or forwards
- Insert: add links to include new node in chain

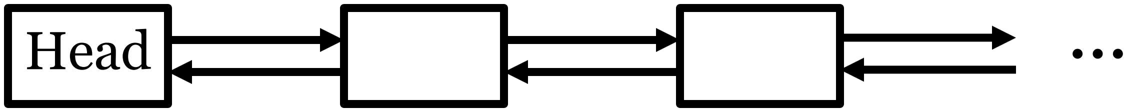


  - If sorted, scan to find insertion position
- Delete: reroute links, then delete victim

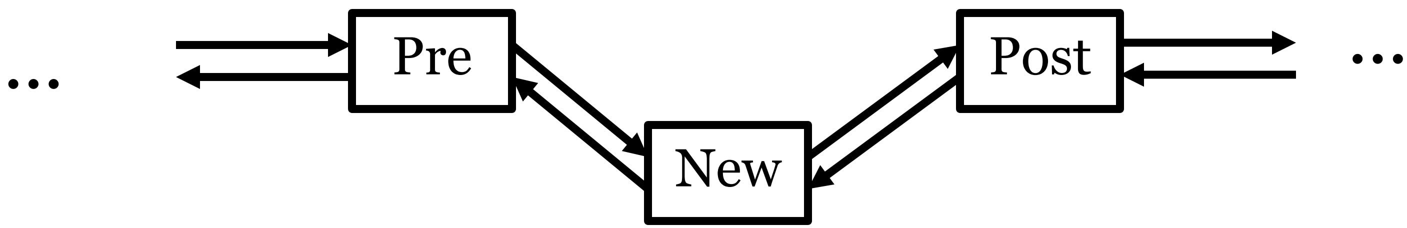


Linked list operations

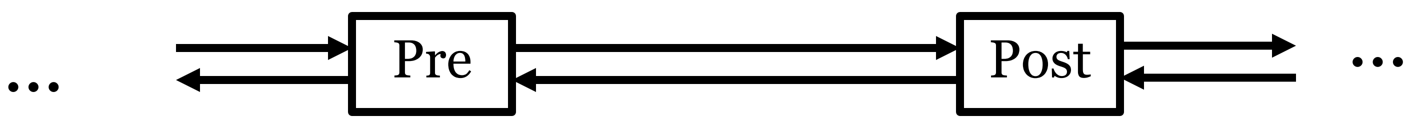
- Search: linear scan



  - If sorted, stop when values are too large
  - If DLL, you can search backwards from end or forwards
- Insert: add links to include new node in chain



  - If sorted, scan to find insertion position
- Delete: reroute links, then delete victim



  - If SLL, need to scan to find previous node

Summary: link-based dictionaries

Operation	Unsorted SLL	Unsorted DLL	Sorted SLL	Sorted DLL
Search(x)	*O(n)*	*O(n)*	*O(n)*	*O(n)*
Delete(x)	*O(n)*	*O(1)*	*O(n)*	*O(1)*
Insert(x)	*O(1)*	*O(1)*	*O(n)*	*O(n)*
Build	n/a	n/a	*O(n lg n)*	*O(n lg n)*
Min()	*O(n)*	*O(n)*	*O(1)*	*O(1)*
Max()	*O(n)*	*O(n)*	*O(1)*	*O(1)*
Predecessor(x)	*O(n)*	*O(n)*	*O(n)*	*O(1)*
Successor(x)	*O(n)*	*O(n)*	*O(1)*	*O(1)*

- **Note:** DLL time is strictly better, asymptotically
  - Trade-off: more space, more pointer manipulation

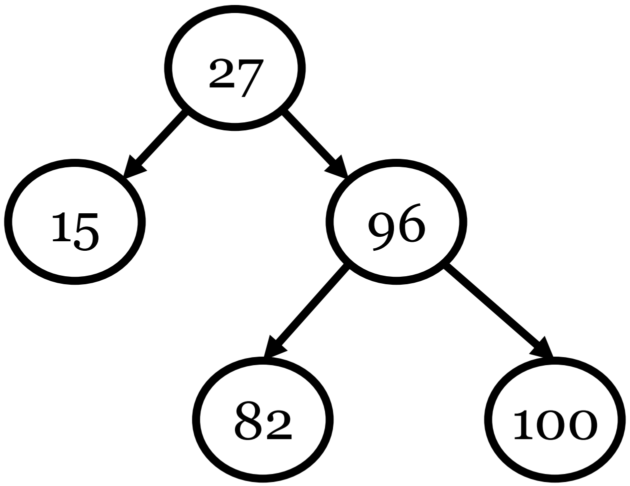
Summary: linear dictionaries

Operation	Unsorted array	Unsorted DLL	Sorted array	Sorted DLL
Search(x)	*O(n)*	*O(n)*	*O(lg n)*	*O(n)*
Delete(x)	*O(1)*	*O(1)*	*O(n)*	*O(1)*
Insert(x)	*O(1), amortized*	*O(1)*	*O(n)*	*O(n)*
Build	n/a	n/a	*O(n lg n)*	*O(n lg n)*
Min()	*O(n)*	*O(n)*	*O(1)*	*O(1)*
Max()	*O(n)*	*O(n)*	*O(1)*	*O(1)*
Predecessor(x)	*O(n)*	*O(n)*	*O(1)*	*O(1)*
Successor(x)	*O(n)*	*O(n)*	*O(1)*	*O(1)*

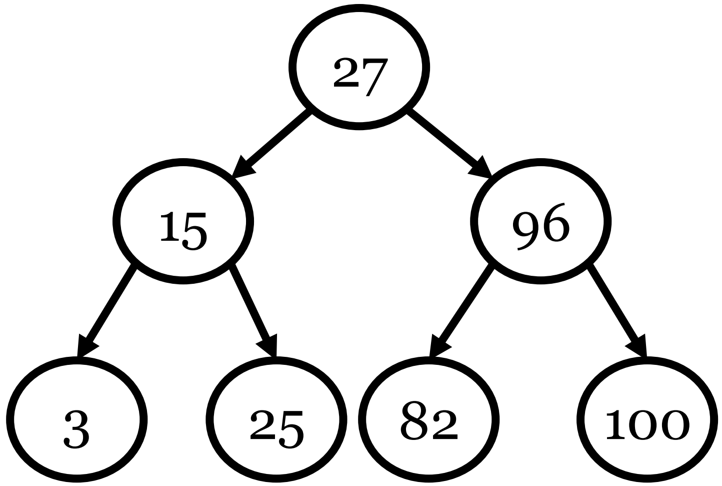
- Arrays are usually preferred, due to lower coefficients

Binary search trees

- Non-linear linked data structure
- Trees start with a *root* node
  - Usually depicted at top
- Each node has two children
  - Use NIL link if no child on left/right
- Nodes also generally store *parent* pointer
  - NIL for root
- **Binary Search Tree Property**
  - All children of left child are equal or smaller
  - All children in right child are equal or larger



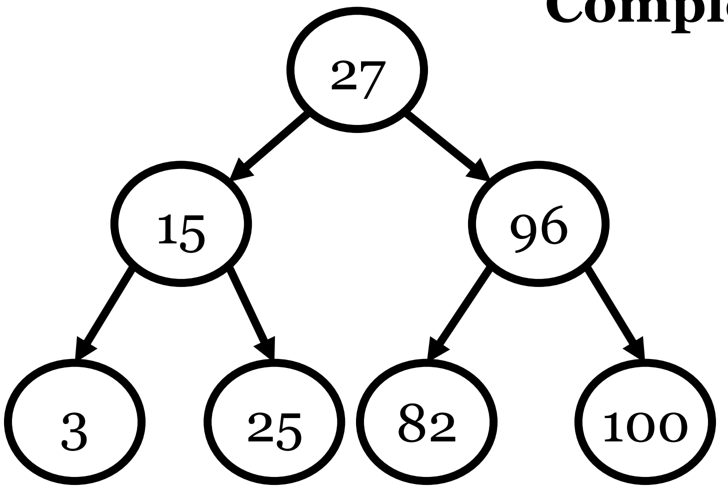
Binary tree lingo



- *Leaf*: node with no children
- *Level*: number of links away from the root
- *Height*: the max level in the tree
- *Complete*: every node above last level has two children
- *Left/right subtree* of a node: tree rooted at node's left/right child
  - Trees are *recursive* data structures
- *Balanced*: each node's left and right subtrees are similar size
- *Degenerate*: BST with only left or right children

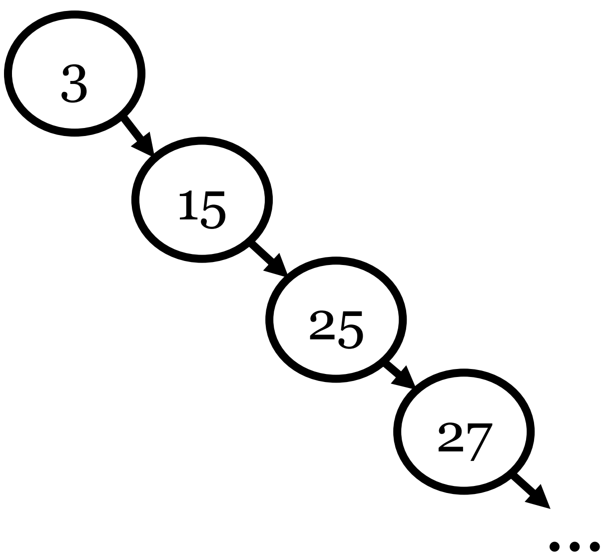
BST properties

Complete BSTs



- Level i has 2^i nodes
- Total nodes: $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
- Height: $O(\lg n)$
- Leaves: approx. $n / 2$
- Root contains median element

Degenerate BSTs



- Every level has 1 node
- Total nodes: n
- Height: $O(n)$
- Leaves: 1
- Essentially a sorted linked list
  - Left children: descending
  - Right children: ascending

BST dictionary operations

- **Search(x)**
  - Binary search
  - Start with Search(root, x)
  - *O(1)* time per call
  - *Worst case: h* calls (height of BST)
  - *O(h) time*

```
Algorithm: Search(node, x)
if node = NIL then
 | return NIL;
else if node.data = x then
 | return node;
else if node.data > x then
 | return Search(node.left, x);
else
 | return Search(node.right, x);
end
```

- **Insert(x)**
  - Binary search
  - Insert as root or call Insert(root, x)
  - *O(h) time*

```
Algorithm: Insert(node, x)
if x ≤ node.data then
 | if left = NIL then
 | | left = NewNode(x);
 | else
 | | Insert(left, x);
 | end
else
 | if right = NIL then
 | | right = NewNode(x);
 | else
 | | Insert(right, x);
 | end
end
```

BST dictionary operations

- **Delete(x)**
  - Binary search
  - Special cases depending on children
    - 0 children: delete
    - 1 child: replace w/ child
    - 2 children: find right ST min, swap
  - Worst case analysis
    - *O(h)* to find *x*
    - *O(h)* to find RST min
    - *O(h) time*

```
Algorithm: Delete(node, x)
if node = NIL then
    return;
else if node.data > x then
    Delete(node.left, x);
else if node.data < x then
    Delete(node.right, x);
else
    if node.left = NIL and node.right = NIL then
        Set node.parent's child pointer to NIL;
        free node;
    else if node.left ≠ NIL and node.right ≠ NIL
    then
        sub = min(node.right);
        Remove sub.parent's child link;
        Set sub's 3 links to match node;
        Set sub's parent's and child's links to sub;
        free node;
    else if node.left ≠ NIL then
        Set parent's child pointer to node.right;
        node.right.parent = node.parent;
        free node;
    else
        Set parent's child pointer to node.left;
        node.left.parent = node.parent;
        free node;
    end
end
```

Balanced Binary Search Trees

- How tall are BSTs?
  - Best case: $O(\lg n)$
  - Average case: $O(\lg n)$
  - Worst case: $O(n)$

- Balanced BSTs
  - Sophisticated variants of BST
  - *Guarantee* $O(\lg n)$ height with constant overhead
    - Red-Black trees, AVL trees, etc.
  - We are not going to cover details of Balanced BSTs

Summary: BST dictionaries

Operation	Binary Search Tree	Balanced BST	Unsorted array	Sorted array
Search(x)	*O(h)*	*O(lg n)*	*O(n)*	*O(lg n)*
Delete(x)	*O(h)*	*O(lg n)*	*O(1)*	*O(n)*
Insert(x)	*O(h)*	*O(lg n)*	*O(1)**	*O(n)*
Build	*O(n lg n)*	*O(n lg n)*	n/a	*O(n lg n)*
Min()	*O(h)*	*O(lg n)*	*O(n)*	*O(1)*
Max()	*O(h)*	*O(lg n)*	*O(n)*	*O(1)*
Predecessor(x)	*O(h)*	*O(lg n)*	*O(n)*	*O(1)*
Successor(x)	*O(h)*	*O(lg n)*	*O(n)*	*O(1)*

- **Advantage:** *O(lg n)* is *much* better than *O(n)* for large data
- **Disadvantage:** *O()* hides larger coefficients for BSTs

Hash tables

- Sparse array-based data structure
- Insert elements according to a *hash function*
  - Function that maps elements in domain to integers 0 to size of array minus one (m-1)
  - Must take *O(1)* time
- **Example hash function**
  - $f : \mathbb{Z} \to [0, m-1]$
  - $f(x) = x \bmod m$
  - Most hash functions use modulus to ensure range
- **Hash table example**
  - Size = 10, hash function: mod 10
  - Inserting 3, 15, 27, 82, 96, 100

100		82	3		15	96	27		

Collisions

- What do we do when two values map to the same location?

100		82	3		15	96	27		

25

- Insert 25
- Two basic solutions
- Separate chaining
  - Each location is the head of a linked list
  - Append new element to list
  - Never "need" to reallocate
- Open addressing
  - Find the next open location, insert there
    - Can scan quadratically to avoid "congestion"
  - No links, so table can be larger with same memory
  - Deleting an element requires reinserting everything that follows
- Both potentially require scanning to find element

Operations

- **Search(x)**
  - Hash element
  - Scan linked list (or until empty location)
  - Worst case: *O(n)*
- **Insert(x)**
  - Hash element
  - Append to linked list (or scan for open location)
  - Worst case: *O(1)* (or *O(n)*)
- **Delete(x)**
  - Hash element
  - Delete from linked list (or scan/delete/re-insert)
  - Worst case: *O(n)* (or $O(n^2)$)

```
Algorithm: Search(x)
loc = Hash(x);
return table[loc].Search(x);
```

```
Algorithm: Insert(x)
ins = NewNode(x);
loc = Hash(x);
ins.next = table[loc];
table[loc] = ins;
```

```
Algorithm: Delete(x)
loc = Hash(x);
node = table[loc];
if node.value = x then
    table[loc] = node.next;
    free node;
else
    while node.next ≠ NIL do
        next = node.next;
        if next.value = x then
            node.next = next.next;
            free next;
        node = node.next;
    end
end
```

Hash table complexity

Operation	Separate chaining	Open addressing	Balanced BST
Search(x)	$O(n)$	$O(n)$	$O(\lg n)$
Delete(x)	$O(n)$	$\boldsymbol{O(n^2)}$	$O(\lg n)$
Insert(x)	$O(n)$	$O(n)$	$O(\lg n)$
Build	$\boldsymbol{O(m + n^2)}$	$\boldsymbol{O(m + n^2)}$	$O(n \lg n)$
Resize	$\boldsymbol{O(m + n^2)}$	$\boldsymbol{O(m + n^2)}$	*n/a*
Min()	$O(m + n)$	$O(m)$	$O(\lg n)$
Max()	$O(m + n)$	$O(m)$	$O(\lg n)$
Predecessor(x)	$O(m + n)$	$O(m)$	$O(\lg n)$
Successor(x)	$O(m + n)$	$O(m)$	$O(\lg n)$

- This is awful!
- Why would anyone ever use a hash table?

Why would anyone use a hash table?

- Bad worst-case complexity but great *expected-case* complexity
- Expected-case assumptions
  - Hash function produces $O(1)$ collisions
    - Each inserted value has $O(1)$ duplicates
  - $m = O(n)$

- Search(x)
  - Hashing and scanning take $O(1)$ time
- Insert(x)
  - Hashing and scanning take $O(1)$ time
- Delete(x)
  - Hashing and scanning take $O(1)$ time
  - Reinsertion takes $O(1)$ time (open addressing)

Expected-case complexity

Operation	Separate chaining	Open addressing	Balanced BST
Search(x)	*O(1)*	*O(1)*	*O(lg n)*
Delete(x)	*O(1)*	*O(1)*	*O(lg n)*
Insert(x)	*O(1)*	*O(1)*	*O(lg n)*
Build	*O(n)*	*O(n)*	*O(n lg n)*
Resize	*O(1), amortized*	*O(1), amortized*	*n/a*
Min()	*O(n)*	*O(n)*	*O(lg n)*
Max()	*O(n)*	*O(n)*	*O(lg n)*
Predecessor(x)	*O(n)*	*O(n)*	*O(lg n)*
Successor(x)	*O(n)*	*O(n)*	*O(lg n)*

- This is amazing!
- The three most important techniques are hashing, hashing, and hashing.

-Udi Manber, Chief Scientist, Yahoo! (2001)

Coming up

- Bit vectors
- Non-dictionary data structures

- **Project 1** will be due next Tuesday
  - Sorting algorithms, Big-Oh analysis
- **Exam 1** will be returned Thursday

- **Recommended readings:** Sections 3.8-3.9
- **Practice problems:** 1-2 problems from "Trees and Other Dictionary Structures"