CS231 Algorithms **Handout #19**
Prof Lyn Turbak **March 8, 2001**
Wellesley College

DYNAMIC SETS

Reading: CLR Chapters 11 & 13

Terminology

We will be studying collections of objects. In this context, an **object** is a record with a **key** field and **satellite data** fields. Every object is identified by a **pointer**; when we say that a function takes an object as an argument or returns one as a result, it is manipulating a pointer to the object. In most data structures, some of the satellite data fields are pointers to other objects in the data structure. The null pointer, or **nil**, is a distinguished pointer that stands for the absence of an object; it is used to represent empty lists or trees. We use the term **leaf** to refer to an empty tree and **fringe node** to refer to a tree node whose children are all leaves.

On the next page are pictures showing objects in the context of some of the data structures we will be studying. Each data structure is assumed to be represented by an **entry point record** with fields that refer to objects as well as auxiliary information. For example, an array has an **elements** field and a **length** field; linked and doubly-linked lists have a **head** field that points to the first object in the list; and trees have a **root** field that points to the root of the tree.

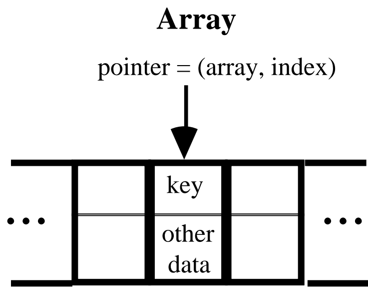
We will assume that new objects are created as follows:

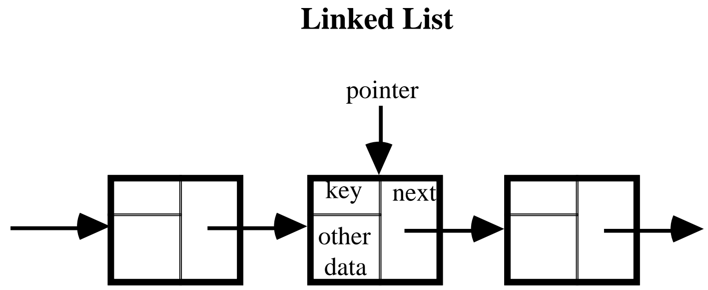
new `Array(n)`
Creates a new array with n slots indexed from 1 to n.

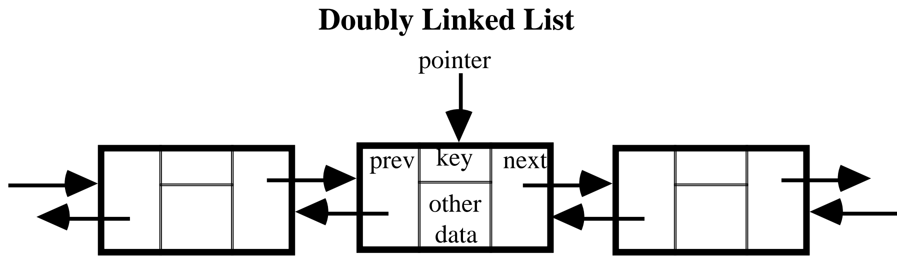
new `ListNode()`
Creates a new list node with fields `key`, `data`, and `next`.

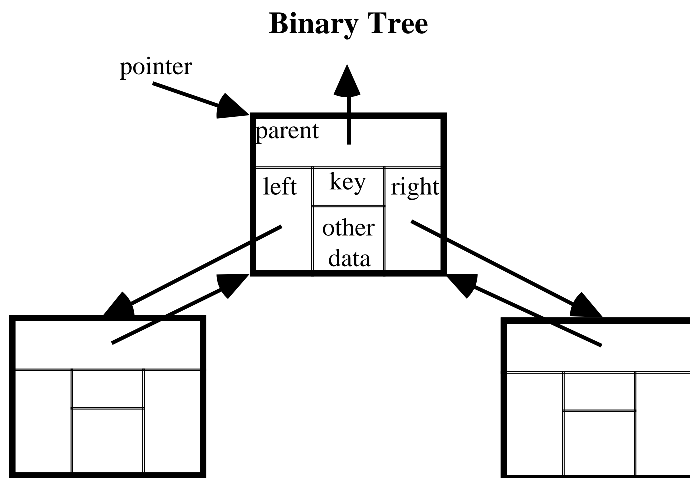
new `DoublyLinkedListNode()`
Creates a new doubly-linked list node with fields `key`, `data`, `next`, and `prev`.

new `BinaryTreeNode()`
Creates a new binary tree node with fields `key`, `data`, `left`, `right`, and `parent`.


Array
pointer = (array, index)
key
other data
• • •
• • •


Linked List
pointer
key
next
other data


Doubly Linked List
pointer
prev
key
next
other data


Binary Tree
pointer
parent
left
key
right
other data

Dynamic Sets

A **dynamic set** is an abstract data type for a mutable collection of objects that supports the following seven operations below. Assume that object keys are **distinct** and are related by a **total order**: i.e., any two keys are related by one of <, = , or >. The distinctness restriction is not essential but simplifies the definitions (e.g., without distinctness, we can't refer to *the* maximal object or *the* object with the next largest key).

Below is the contract for the dynamic set data type. Note that `Insert`, `Delete`, `Predecessor`, and `Successor` take an object pointer as their second argument, not a key. However, it is easy to make versions of these that take keys instead of objects by first using `Search` to find an object with a given key.

`Search(set, key)`
Returns a pointer to an object in `set` with the specified key, or `nil` if there is no such object in `set`.

`Insert(set, obj)`
Destructively updates `set` to add the object `obj`. Insertion may change the fields of `obj` that relate it to other objects in the collection.

`Delete(set, obj)`
Destructively updates `set` to remove the object `obj`.

`Mininum(set)`
Returns the object in `set` with the smallest key.

`Maximum(set)`
Returns the object in `set` with the largest key.

`Predecessor(set, obj)`
Returns the object in `set` whose key directly precedes that of `obj` in the total order of keys in `set`. Returns `nil` if `obj` is the element with the minimal key.

`Successor(set, obj)`
Returns the object in `set` whose key directly follows that of `obj` in the total order of keys in `set`. Returns `nil` if `obj` is the element with the maximal key.

Asymptotic Worst-Case Running Times for Simple Implementations of Dynamic Sets

In the following table, use *n* to refer to the number of elements in the set and *h* to refer to the height of the tree-based representations.

	Search	Insert	Delete	Maximum	Minimum	Successor	Predecesso
Unsorted Array							
Sorted Array							
Unsorted Linked List							
Sorted Linked List							
Unsorted Doubly-Linked List							
Sorted Doubly-Linked List							
Binary Tree							
Binary Search Tree							

Binary Search Trees

A binary search tree (BST) is a binary tree satisfying the **binary search tree property**:

If `left-obj` is in the left subtree of `obj`, `key[left-obj]   key[obj]`
If `right-obj` is in the right subtree of `obj`, `key[right-obj]   key[obj]`

There are many binary search trees corresponding to a given set of elements.
E.g., what are the binary search trees for {A, B, C}?

The objects in a binary search tree can be enumerated in sorted order by inorder traversal of the tree starting at its root:

```
Inorder-Traverse(node, function)
  if node   nil then
    Inorder-Traverse(left[node], function)
    function(node)
    Inorder-Traverse(right[node], function)
```

This gives rise to the `Tree-Sort` sorting algorithm:

Step 1: Insert all objects into a binary search tree.
Step 2: Enumerate the elements of the binary search tree via an in-order traversal.

BST Operations I

```
{This algorithm is recursive, but it's easy to make it iterative.}
BST-Search(node, key)
  if (node = nil) or (key = key[node])
    then return node
  if key < key[node]
    then return BST-Search(left[node], key)
    else return BST-Search(right[node], key)

BST-Minimum(node)
  while left[node]   nil
    do node <- left[node]
  return node

{BST-Maximum is symmetric with BST-Minimum}

{The successor is
  (1) The minimum of the right subtree (if it exists)
  (2) The first leftward parent (if it exists)}
BST-Successor(node)
  if right[node]   nil
    then return BST-Minimum(right[node])
  parent-node <- parent[node]
  child-node <- node
  while (parent-node   nil) and (child-node = right[parent-node])
    do child-node <- parent-node
       parent-node <- parent[parent-node]
  {At this point, either
     (1) parent-node is nil
     (2) child-node = left[parent-node]}
  return parent-node

{BST-Predecessor is symmetric with BST-Successor}
```

BST Operations II

```
BST-Insert(tree, node)
  previous <- nil
  current <- root[tree]
  while current   nil
    do previous <- current
       if key[node] < key[current]
          then current <- left[current]
          else current <- right[current]
  {current is now nil}
  parent[node] <- previous
  if previous = nil
    then root(tree) <- node
    else if key[node] < key[previous]
              then left[previous] <- node
              else right[previous] <- node

{Different from CLR version.
 Assumes there is a dummy header node at root.}
BST-Delete(tree, node)
  if (left[node] = nil) or (right[node] = nil)
    {Case 1: at least one child of node is nil}
    then if node = left[parent[node]]
           then left[parent[node]] <- Single-Child(node)
           else right[parent[node]] <- Single-Child(node)
    {Case 2: both children of node are non-nil.}
    else succ <- BST-Successor(tree, node)
         {Delete succ from current position.
          Guaranteed to use Case 1.}
         BST-delete(tree, succ)
         {Splice succ into position of node}
         parent[left[node]] <- succ
         left[succ] <- left[node]
         parent[right[node]] <- succ
         right[succ] <- right[node]
         parent[succ] <- parent[node]
         if node = left[parent[node]]
           then left[parent[node]] <- succ
           else right[parent[node]] <- succ

{If one child is nil, returns the other one.
 Returns nil if both children are nil.}
Single-Child(node)
  if left[node] = nil
    then return right[node]
    else return left[node]
```

Notes

All seven BST operations have (h) worst-case running times, where h is the height of the tree. What is the relationship between the height h and the number of elements n?

Best case:

Worst case:

Average case:

For the average case, assume that trees are formed by sequentially inserting the elements of a randomly permuted array of n objects into an initially empty BST. Note that there is a close correspondence between the resulting tree and the partitionings of quick sort.