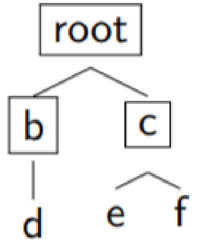
Binary Search Trees

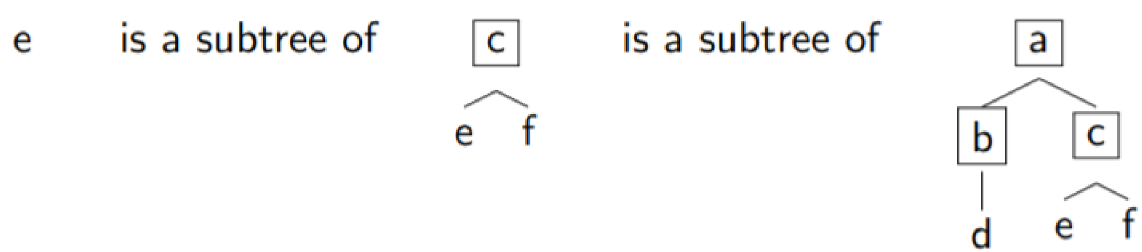
Definitions

- A *tree* is a collection of *nodes* and directed edges such that
  - Each node has a unique parent node, except the root node, which has no parent
  - Each node has a (possibly empty) set of children
  - Parent and child nodes are connected by a directed edges from parent to child
  - There is a unique path from the root to each node in the tree



Definitions

- A *leaf* is a node that has no children
- An *internal node* is a node that has at least one child
- Two nodes with the same parent are called siblings
- *Grandparent, ancestor, descendant* etc. are defined analogously
- A *subtree* consists of a node along with *all* its descendants



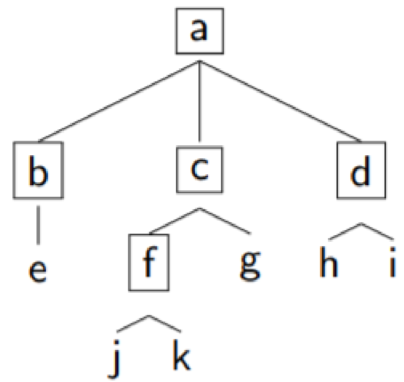
Definitions

- A *path* from node n_0 to node n_k is a sequence $n_0, n_1, \dots, n_k$ such that for all i, n_i is the parent of n_{i+1}
- The *length* of the path is the number of *edges* it contains
  - Not the number of nodes (usually)
- For each node *n* the *depth* of n is the length of the unique path from the root to *n*
- The *height* of *n* is the length of the longest path from *n* to a leaf
- The *height* of a tree is equal to the height of the root node of a the tree

Definitions

- The *branching factor* of a tree is the maximum number of children in any of its nodes
- A *binary tree* is a tree with a branching factor 2
- A *full node* has the maximum number of children
- A *level* in a tree is the set of all nodes in the tree at a given depth
- A *complete tree* is one where all levels are full except the bottom level, which has been filled from left to right
  - Heaps are complete trees
- A *full tree* is a complete tree whose last level has been filled completely

Example



- The path c-f-k has length 2
- The (depth of f) = (depth of g) = 2
- (height of f) = 1, (height of g) = 0
- (depth of a) = 0
- The height of the tree is 3

Traversal

- Want to visit all the nodes in the tree in some order, and apply some operation to each node

preorder traversal pseudocode

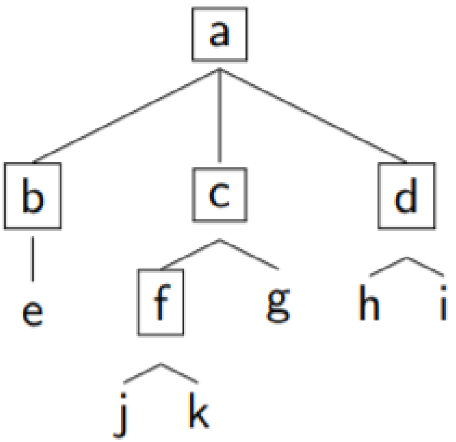
```
void pre_order_traversal(node)
{
    visit(node);
    for each child of node {
        pre_order_traversal(child);
    }
}
```

postorder traversal pseudocode

```
void post_order_traversal(node)
{
    for each child of node {
        post_order_traversal(child);
    }
    visit(node);
}
```

preorder traversal pseudocode

```
void pre_order_traversal(node)
{
    visit(node);
    for each child of node {
        pre_order_traversal(child);
    }
}
```

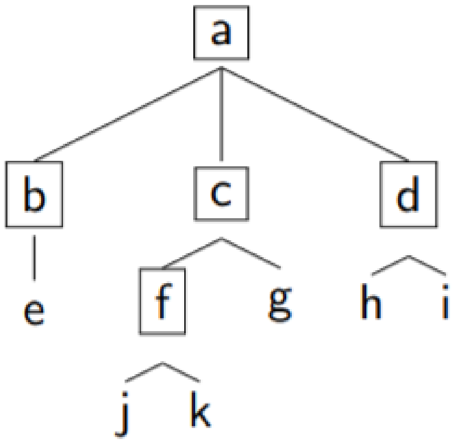


Pre-order traversal (aka depth-first traversal): a, b, e, c, f, j, k, g, d, h, I

- Idea: visit node, then visit each of its subtrees. When visiting a subtree, start from the root

postorder traversal pseudocode

```
void post_order_traversal(node)
{
    for each child of node {
        post_order_traversal(child);
    }
    visit(node);
}
```

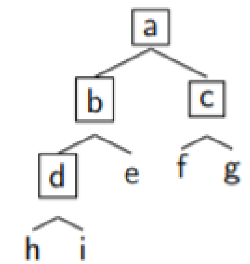


Post-order: e, b, j, k, f, g, c, h, I, d, a

Visit each of the subtrees of the node, then visit the node itself

- For binary trees (each node has at most two children): *in-order traversal*

```
void in_order_traversal(node)
{
    if (node->left)   in_order_traversal(node->left);
    visit(node);
    if (node->right)  in_order_traversal(node->right);
}
```



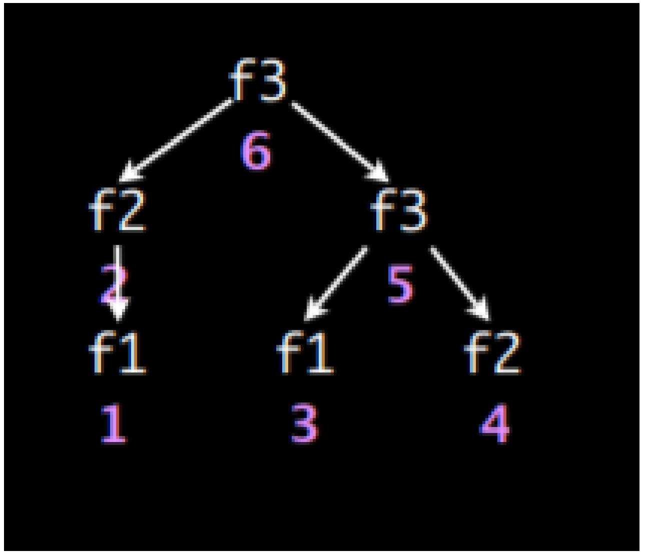
In-order traversal: h, d, I, b, e, a, f, c, g

Visualizing the call stack

```
int x = f3(f2(f1()), f3(f1(), f2(5)+1)
           6  2  1     5  3       4
```

We can construct a tree where f2 is the child of f3 is the call to f2 is needed to evaluate f3, etc.

Traverse the tree in postorder to determine the order of calls



Tree implementation

```
typedef struct bin_tree_node {
    const void *data;
    struct bin_tree_node *left;
    struct bin_tree_node *right;
} bin_tree_node_t;

typedef struct bin_tree {
    bin_tree_node_t *root;
    size_t size;           /* not strictly necessary */
} bin_tree_t;
```

Searching a tree

- Want to find a node with a specific value in the tree
- *n* nodes
- Worst-case run-time: O(n)

```
search(root, value)
  if root = NULL
     return NULL
  if root->data = value
     return root
  left = search(root->left)
  if(left != NULL)
    return left
  return search(root->right)
```

```
bool search(const bin_tree_node_t *root, const void *value)
{
    return root != NULL &&
        (root->data == value ||         /* think about this... */
         search(root->left, value) ||
         search(root->right, value));
}
```

Binary Search Trees (BST)

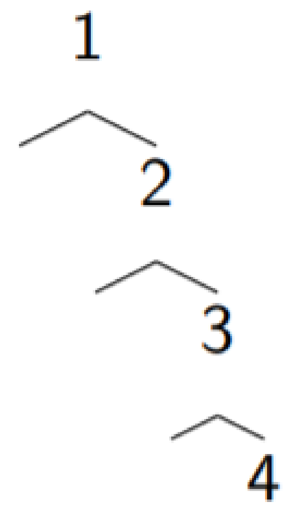
- A tree T where
  - All values in T are *comparable* (can be ordered)
  - All nodes in the left subtree of a node with value *k* have a value less than *k*
  - All nodes in the right subtree of a node with value *k* have a value greater than *k*
  - (Implication: all values are different)

Searching a BST

```
search(root, value)
  if root = NULL
     return NULL
  if value = root->data
     return root
  else if value < root->data
    return search(root->left, value)
  else
    return search(root->right, value)
```

BST search run-time

- In the worst case, O(n)
- Need to traverse all nodes
- If height h is known, runtime is O(h)
- If the tree is complete, the height is approx. log(n), so the runtime is O(log(n))



BST implementation

- Use the same structure as for an ordinary binary tree, but add a comparator function

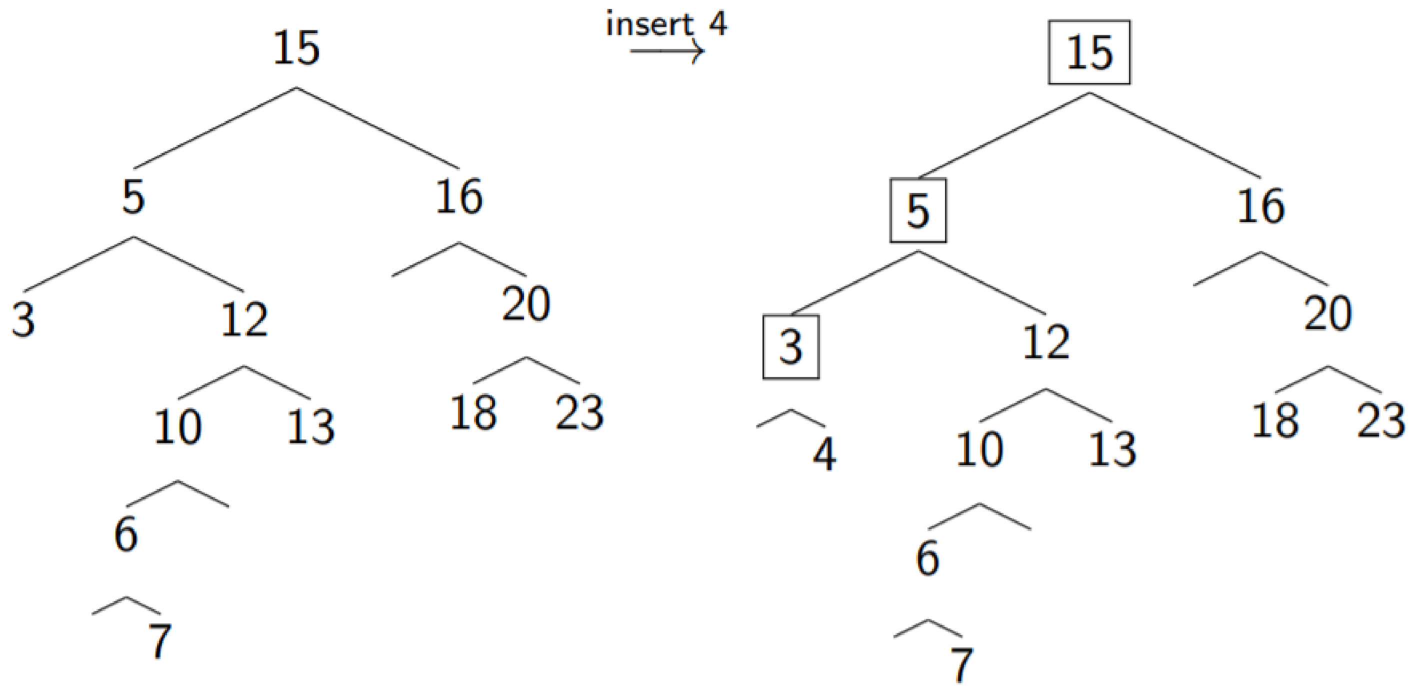
```
typedef struct bst_node {
    const void *data;
    struct bst_node *left;
    struct bst_node *right;
} bst_node_t;

struct bst {
    bst_node_t *root;
    size_t size;
    int (*cmp)(const void *, const void *);
};
```

Inserting into a BST

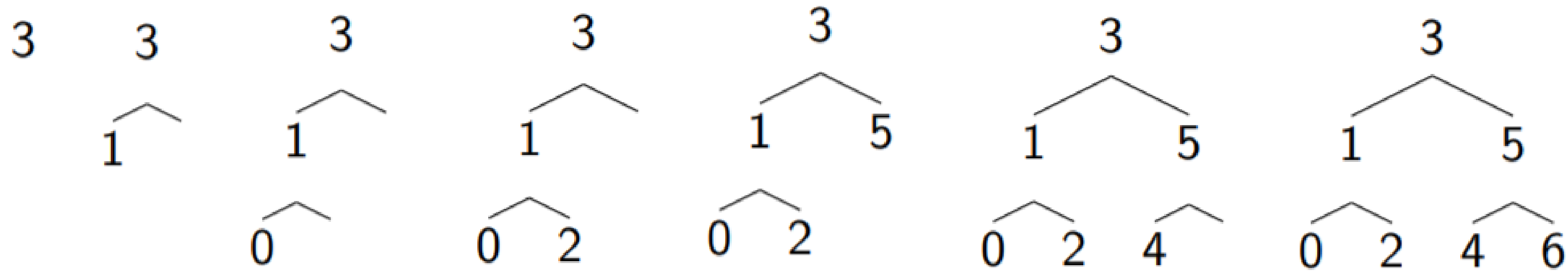
- Need to insert a new node into the BST, and want for the resulting tree to still be a BST

```
insert(cur_node, value)
  if cur_node = NULL
    create a node with data=value, insert it instead of cur_node
  if value < cur_node->data
    insert(cur_node->left, value)
  else
    insert(cur_node->right, value)
```


15
5
16
3
12
20
10
13
18
23
6
7
insert 4
15
5
16
3
12
20
4
10
13
18
23
6
7

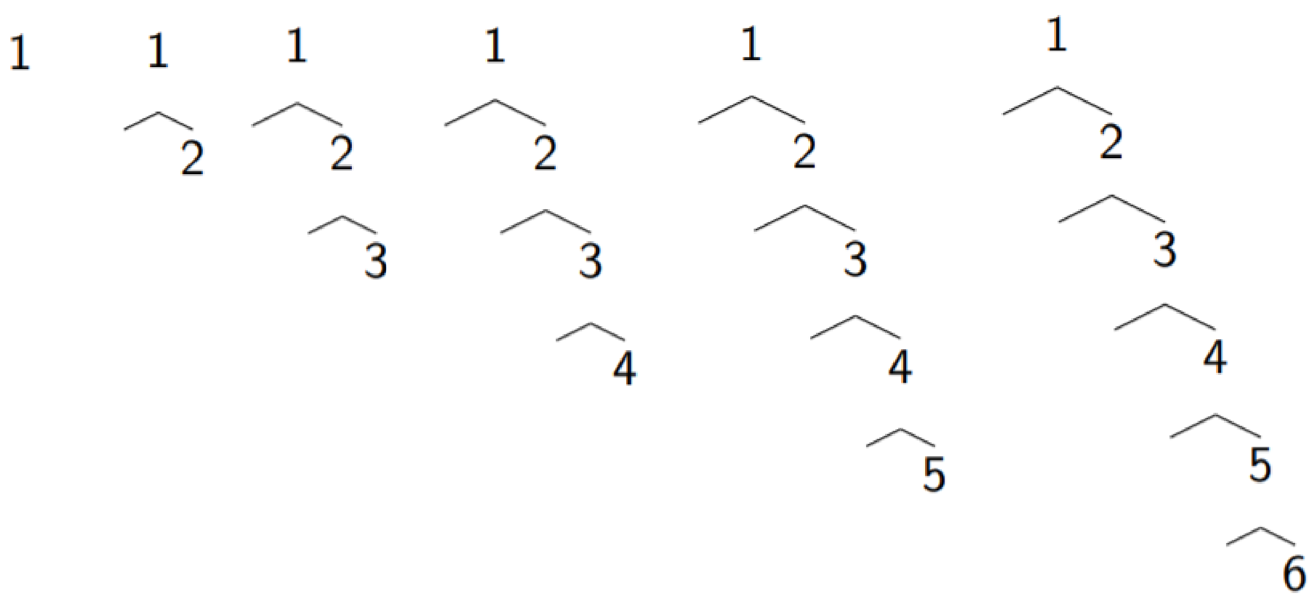
Insertion order matters

- Insert 3, 1, 0, 2, 5, 4, 6



Insertion order matters

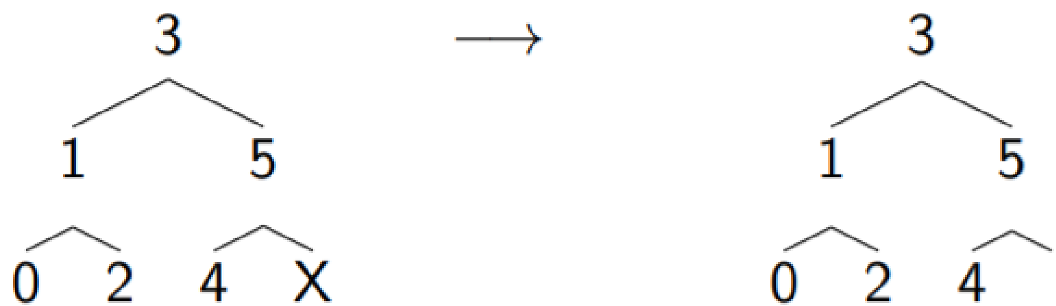
- Insert 1, 2, 3, 4, 5, 6



Deleting from a BST

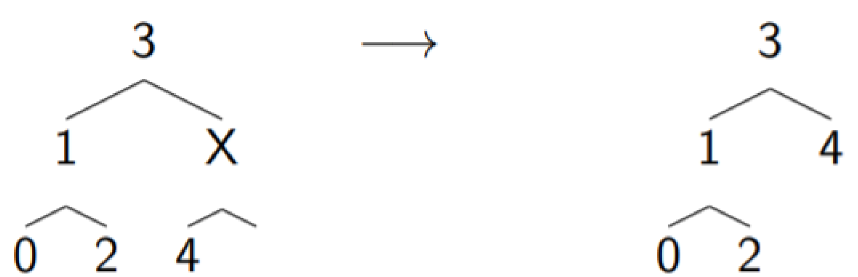
- Plan:
  - Find the node to delete
  - Delete it
- Cases of node deletion
  - Delete a node with no children (easy)
  - Delete a node with one child
  - Delete a node with two children

Deleting a node with no children



Deleting a node with one child

- Replace the node with its child
- The child has the same properties as the parent with respect to the grandparent – if the parent is smaller than the grandparent, so is the child; if the parent is larger than the grandparent, so is the child



Deleting a node with two children

- Leave the node in the tree
- Replace the value stored in the node with the successor value
  - Successor: the next larger value in the tree
- Idea: the successor is larger or equal to the left child, and smaller or equal to the right child

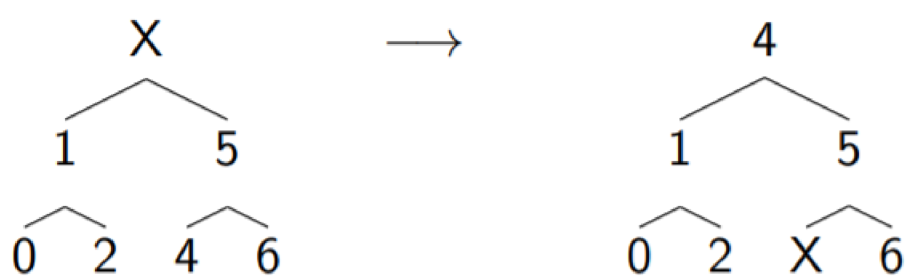
Finding the successor

```
successor(node)
  cur_node = node->right
  while cur_node->left != NULL
    cur_node = cur_node->left
  return cur_node
```

- Idea: look in the right subtree, and then go left as much as possible
  - Need the smallest node in the right subtree

Deleting a node with two childen

- Find the successor
- Swap the node with its successor
- Delete the node from its new location
  - The successor has at most one child, since it definitely doesn't have a left subtree
    - We went left as much as possible



BST Implementation

- Bag ADT
  - A collection of elements
  - Want to make the ADT independent of the implementation

```
bag_t *bag_create(int (*cmp)(elem_t, elem_t));

void bag_destroy(bag_t *b);

size_t bag_size(const bag_t *b);

bool bag_contains(const bag_t *b, elem_t e);

bool bag_insert(bag_t *b, elem_t e);

bool bag_remove(bag_t *b, elem_t e);
```

BST implementation of Bag

In bst_bag.c and not bag.h since those structs are only used internally

```
/* TYPE bst_node_t -- The type of tree nodes used to store elements. */
typedef struct bst_node {
    elem_t elem;
    struct bst_node *left;
    struct bst_node *right;
} bst_node_t;

/* TYPE bag -- Definition of struct bag from the header file. */
struct bag {
    size_t size;
    bst_node_t *root;
    int (*cmp)(elem_t, elem_t);
};
```

Bag functions call BST functions

(not declared in bag.h)

```
void bst_destroy(bst_node_t *root)
{
    if (root) {
        // Standard post-order traversal to free all the memory.
        bst_destroy(root->left);
        bst_destroy(root->right);
        free(root);
    }
}
```

(declared in bag.h)

```
void bag_destroy(bag_t *b)
{
    // Recursively free the memory for tree nodes, then free b itself.
    bst_destroy(b->root);
    free(b);
}
```

Bag functions call BST functions

```
bool bst_remove(bst_node_t **root, elem_t elem, int (*cmp)(elem_t, elem_t))
{
    // This function takes a *pointer* to the pointer to the root node, so that
    // we can modify its value directly -- for example, when this is called with
    // argument &b->root, it gets the *address* of the pointer b->root, so that
    // if b->root->elem == elem and b->root is the only node, we can change the
    // value of b->root to become NULL directly from here.
    if (! *root)
        return false;
    else if (cmp(elem, (*root)->elem) < 0)
        return bst_remove(&(*root)->left, elem, cmp);
```

```
bool bag_remove(bag_t *b, elem_t e)
{
    // Remove the element recursively, starting at the root.
    if (bst_remove(&b->root, e, b->cmp)) {
        b->size--;
        return true;
    } else {
        return false;
    }
}
```

remove_min

- If root doesn’t have a left child, it is the minimum
  - Replace the root with its right child
- If root has a left child, remove_min(left child)

```
elem_t bst_remove_min(bst_node_t **root)
{
    // As above, this function takes a pointer to a pointer to the root node, so
    // that the value of the pointer can be modified directly from in here.
    if ((*root)->left) {
        /* *root is not the minimum, keep going */
        return bst_remove_min(&(*root)->left);
    } else {
        /* remove *root */
        bst_node_t *old = *root;
        elem_t min = (*root)->elem;
        *root = (*root)->right;
        free(old);
        return min;
    }
}
```

bst_remove

```
bool bst_remove(bst_node_t **root, elem_t elem, int (*cmp)(elem_t, elem_t))
{
    // This function takes a *pointer* to the pointer to the root node, so that
    // we can modify its value directly -- for example, when this is called with
    // argument &b->root, it gets the *address* of the pointer b->root, so that
    // if b->root->elem == elem and b->root is the only node, we can change the
    // value of b->root to become NULL directly from here.
    if (! *root)
        return false;
    else if (cmp(elem, (*root)->elem) < 0)
        return bst_remove(&(*root)->left, elem, cmp);
    else if (cmp(elem, (*root)->elem) > 0)
        return bst_remove(&(*root)->right, elem, cmp);
    else { /* (cmp(elem, (*root)->elem) == 0) */
        // We've found the value to remove; check if *root has two children.
        if ((*root)->left && (*root)->right) {
            (*root)->elem = bst_remove_min(&(*root)->right);
        } else { /* remove *root */
            bst_node_t *old = *root;
            *root = (*root)->left ? (*root)->left : (*root)->right;
            free(old);
        }
        return true;
    }
}
```

Using Bag

```
int float_cmp(const void *a, const void *b)
{
    return *(float *) a < *(float *) b ? -1
         : *(float *) a > *(float *) b; /* ? 1 : 0 redundant */
}
```

```
bag_t *b1 = bag_create(float_cmp);
```

```
bag_insert(b1, &elts[i])
```