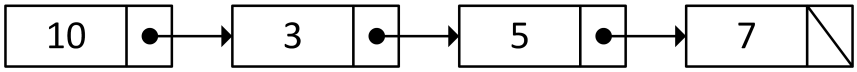
Linked Data Structures

Readings: CP:AMA 17.5

The primary goal of this section is to be able to use linked lists and trees.

Linked lists

Racket's list type is more commonly known as a ***linked list***.

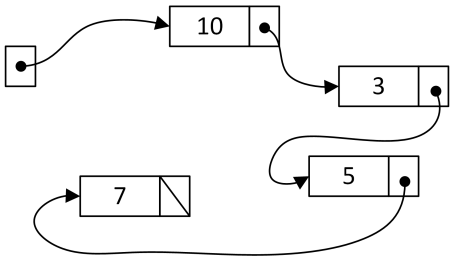


Each ***node*** contains an ***item*** and a **link** (*pointer*) to the ***next*** node in the list.

In C, the *link* in the *last node* is `NULL` pointer.

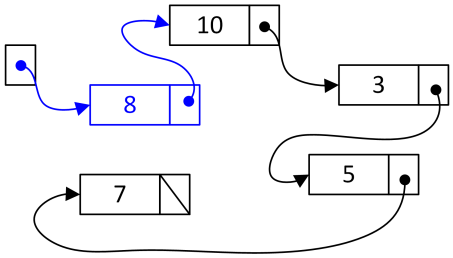
For convenience, we will initally store `int`s in our linked lists.

Linked lists are usually represented as a link (pointer) to the ***front***.



Unlike arrays, linked list nodes are **not** arranged sequentially in memory. There is no fast and convenient way to "jump" to the i-th element. The list must be **traversed** from the *front*. Traversing a linked list is $O(n)$.

A significant advantage of a linked list is that its length can easily change, and the length does not need to be known in advance.



The memory for each node is allocated dynamically (*i.e.,* using *dynamic memory*).

Structure definitions

A `llist` points to the `front` node (which is `NULL` for an empty list).

Each `llnode` stores an `item` and a link (pointer) to the `next` node (which is `NULL` for the last node).

```
struct llnode {
  int item;
  struct llnode *next;
};

struct llist {
  struct llnode *front;
};
```

A C structure can contain a *pointer* to its own structure type. `llnode` is a **recursive data structure** (but `llist` is not).

There is no “official” way of naming or implementing a linked list node in C.

The CP:AMA textbook and other sources use slightly different conventions.

The structure we present here is often called a “wrapper strategy”, because the `llist` structure “wraps” around the front of the list.

Creating a linked list

```
// list_create() creates a new, empty list
// effects: allocates memory

struct llist *list_create(void) {
  struct llist *lst = malloc(sizeof(struct llist));
  lst->front = NULL;
  return lst;
}


int main(void) {

  struct llist *lst = list_create();
  // ...
}
```


Adding and creating nodes

```
// new_node(i, pnext) creates a new linked list node
// effects: allocates memory

struct llnode *new_node(int i, struct llnode *pnext) {
  struct llnode *node = malloc(sizeof(struct llnode));
  node->item = i;
  node->next = pnext;
  return node;
}


// add_front(i, lst) adds i to the front of lst

void add_front(int i, struct llist *lst) {
  lst->front = new_node(i, lst->front);
}
```


Traversing a list

We can *traverse* a list **iteratively** or **recursively**.

When iterating through a list, we typically use a (`llnode`) pointer to keep track of the "current" node.

```
int list_length(const struct llist *lst) {
  int len = 0;
  struct llnode *node = lst->front;
  while (node) {
    ++len;
    node = node->next;
  }
  return len;
}
```

Remember (`node`) will be false at the end of the list (`NULL`).

When using **recursion**, remember to recurse on a node (`llnode`) not the list (`llist`).

```
int length_nodes(const struct llnode *node) {
  if (node == NULL) {
    return 0;
  }
  return 1 + length_nodes(node->next);
}


int list_length(const struct llist *lst) {
  return length_nodes(lst->front);
}
```


Destroying a list

In C, we don't have a *garbage collector*, so we must be able to `free` our linked list. We need to free every node and the list itself.

When using an iterative approach, we are going to need *two* node pointers to ensure that the nodes are `free`d in a safe way.

```
void list_destroy(struct llist *lst) {
  struct llnode *curnode = lst->front;
  struct llnode *nextnode = NULL;
  while (curnode) {
    nextnode = curnode->next;
    free(curnode);
    curnode = nextnode;
  }
  free(lst);
}
```


With a recursive approach, it is more convenient to free the *rest* of the list before we `free` the first node.

```
void free_nodes(struct llnode *node) {
  if (node) {
    free_nodes(node->next);
    free(node);
  }
}

void list_destroy(struct llist *lst) {
  free_nodes(lst->front);
  free(lst);
}
```

Duplicating a list

The recursive function is the most straightforward.

```
struct llnode *dup_nodes(const struct llnode *oldnode) {
  if (oldnode == NULL) {
    return NULL;
  }
  return new_node(oldnode->item, dup_nodes(oldnode->next));
}

struct llist *list_dup(const struct llist *oldlist) {
  struct llist *newlist = list_create();
  newlist->front = dup_nodes(oldlist->front);
  return newlist;
}
```

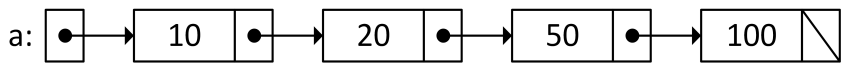

The iterative solution is more complicated:

```
struct llist *list_dup(struct llist *oldlist) {
  struct llist *newlist = list_create();
  if (oldlist->front) {
    newlist->front = new_node(oldlist->front->item, NULL);
    struct llnode *oldnode = oldlist->front->next;
    struct llnode *node = newlist->front;
    while (oldnode) {
      node->next = new_node(oldnode->item, NULL);
      node = node->next;
      oldnode = oldnode->next;
    }
  }
  return newlist;
}
```

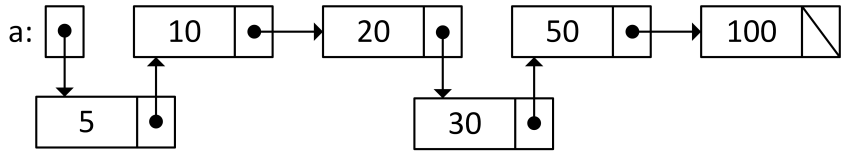

Insert into the “middle”

When inserting into the “middle” of a linked list, we need to know *where* to insert.

Consider inserting into a **sorted** linked list:




```
insert( 5, a);
insert(30, a);
```



```
// insert(i, slst) inserts i into sorted list slst
// requires: slst is sorted
// effects:  modifies slst
// time:     O(n), where n is the length of slst

void insert(int i, struct llist *slst) {
  if (slst->front == NULL || i < slst->front->item) {
    add_front(i, slst);
  } else {
    struct llnode *prevnode = slst->front;
    while (prevnode->next && i > prevnode->next->item) {
      prevnode = prevnode->next;
    }
    prevnode->next = new_node(i, prevnode->next);
  }
}
```


Removing nodes

```
void remove_front(struct llist *lst) {
  assert(lst->front);
  struct llnode *old_front = lst->front;
  lst->front = lst->front->next;
  free(old_front);
}
```

In Racket, the `rest` function does not actually *remove* the `first` element, instead it provides a pointer to the next node.

Removing a node from an arbitrary list position is more complicated.

```
// remove_item(i, lst) removes the first occurrence of i in lst
//   returns true if item is successfully removed

bool remove_item(int i, struct llist *lst) {
  if (lst->front == NULL) return false;
  if (lst->front->item == i) {
    remove_front(lst);
    return true;
  }
  struct llnode *prevnode = lst->front;
  while (prevnode->next && i != prevnode->next->item) {
    prevnode = prevnode->next;
  }
  if (prevnode->next == NULL) return false;
  struct llnode *old_node = prevnode->next;
  prevnode->next = prevnode->next->next;
  free(old_node);
  return true;
}
```

Caching information

Consider that we are writing an application where the `length` of a linked list will be queried often.

Typically, finding the length of a linked list is $O(n)$.

However, we can store (or "cache") the length *in the `llist` structure*, so the length can be retrieved in $O(1)$ time.

```
struct llist {
  struct llnode *front;
  int length;
};
```


Naturally, other list functions would have to update the `length` as necessary:

- `list_create` would initialize length to zero
- `add_front` would increment length
- `remove_front` would decrement length
- *etc.*

Data integrity

The introduction of the `length` field to the linked list may seem like a great idea to improve efficiency.

However, it introduces new ways that the structure can be corrupted.

What if the `length` field does not accurately reflect the true length?

For example, imagine that someone implements the `remove_item` function, but forgets to update the `length` field?

Or a naïve coder may think that the following statement removes all of the nodes from the list.

```
lst->length = 0;
```

Whenever the same information is stored in more than one way, it is susceptible to *integrity* (consistency) issues.

Advanced testing methods can often find these types of errors.

If data integrity is an issue, it is often better to repackage the data structure as a separate ADT module and only provide interface functions to the client.

This is an example of **security** (protecting the client from themselves).

Queue ADT

A queue is like a "lineup", where new items go to the "back" of the line, and the items are removed from the "front" of the line. While a stack is LIFO, a queue is FIFO (first in, first out).

Typical queue ADT operations:

- `add_back`: adds an item to the end of the queue
- `remove_front`: removes the item at the front of the queue
- `front`: returns the item at the front
- `is_empty`: determines if the queue is empty

A Stack ADT can be easily implemented using a dynamic array (as we did in Section 10) or with a linked list.

While it is possible to implement a Queue ADT with a dynamic array, the implementation is a bit tricky. Queues are typically implemented with linked lists.

The only concern is that an `add_back` operation is normally $O(n)$.

However, if we maintain a pointer to the back (last element) of the list, in addition to a pointer to the front of the list, we can implement `add_back` in $O(1)$.

```c
// queue.h

// all operations are O(1) (except destroy)

struct queue;

struct queue *queue_create(void);

void queue_add_back(int i, struct queue *q);

int queue_remove_front(struct queue *q);

int queue_front(struct queue *q);

bool queue_is_empty(struct queue *q);

void queue_destroy(struct queue *q);
```



```c
// queue.c (IMPLEMENTATION)

struct llnode {
  int item;
  struct llnode *next;
};

struct queue {
  struct llnode *front;
  struct llnode *back;      // <--- NEW
};

struct queue *queue_create(void) {
  struct queue *q = malloc(sizeof(struct queue));
  q->front = NULL;
  q->back = NULL;
  return q;
}
```



```c
void queue_add_back(int i, struct queue *q) {
  struct llnode *node = new_node(i, NULL);
  if (q->front == NULL) {
    q->front = node;
  } else {
    q->back->next = node;
  }
  q->back = node;
}

int queue_remove_front(struct queue *q) {
  assert(q->front);
  int retval = q->front->item;
  struct llnode *old_front = q->front;
  q->front = q->front->next;
  free(old_front);
  if (q->front == NULL) {
    q->back = NULL;
  }
  return retval;
}
```

The remainder of the Queue ADT is straightforward.

```
int queue_front(struct queue *q) {
  assert(q->front);
  return q->front->item;
}

bool queue_is_empty(struct queue *q) {
  return q->front == NULL;
}

void queue_destroy(struct queue *q) {
  while (!queue_is_empty(q)) {
    queue_remove_front(q);
  }
  free(q);
}
```

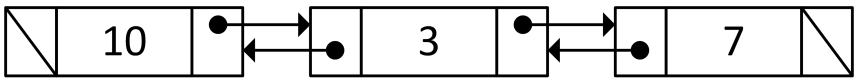

Node augmentation strategy

In a **node augmentation strategy**, each *node* is *augmented* to include additional information about the node or the structure.

For example, a **dictionary** node can contain both a *key* (item) and a corresponding *value*.

Or for a **priority queue**, each node can additionally store the priority of the item.

The most common node augmentation for a linked list is to create a ***doubly linked list***, where each node also contains a pointer to the *previous* node. When combined with a `back` pointer, a doubly linked list can add or remove from the front **and back** in $O(1)$ time.

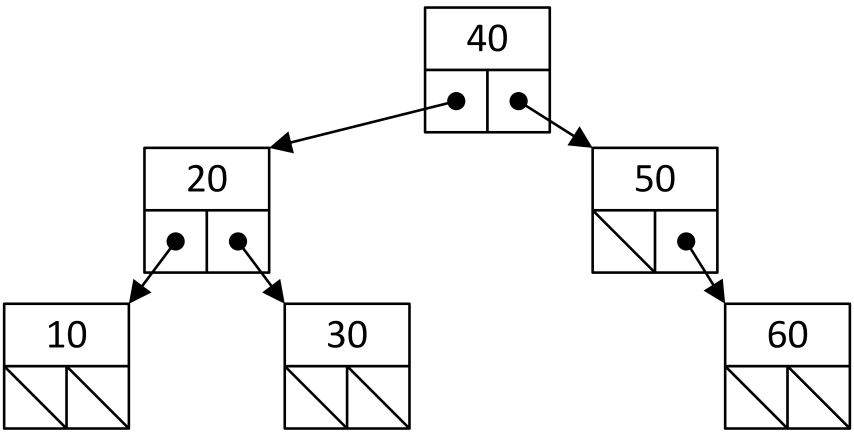


Many programming environments provide a Double-Ended Queue (dequeue or deque) ADT, which can be used as a Stack or a Queue ADT.

Trees

At the implementation level, ***trees*** are very similar to linked lists.

Each node can *link* to more than one node.

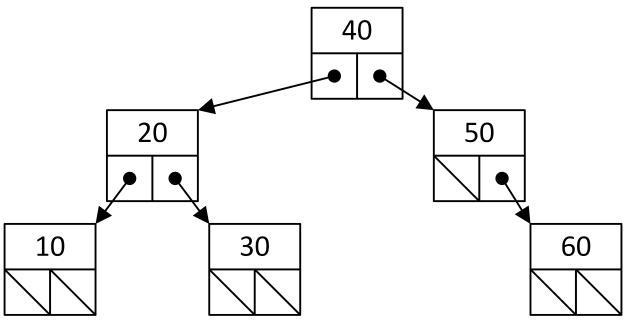


Tree terminology

- the **root node** has no **parent**, all others have exactly one
- nodes can have multiple **children**
- in a **binary tree**, each node has at most two children
- a **leaf node** has no children
- the **height** of a tree is the maximum possible number of nodes from the root to a leaf (inclusive)
- the height of an empty tree is zero
- the number of nodes is known as the **node count**

Binary Search Trees (BSTs)

Binary Search Tree (BSTs) enforce the **ordering property**: for every node with an item i, all items in the left child subtree are less than i, and all items in the right child subtree are greater than i.



Our *BST node* (`bstnode`) is very similar to our linked list node definition.

```
struct bstnode {
  int item;
  struct bstnode *left;
  struct bstnode *right;
};

struct bst {
  struct bstnode *root;
};
```

In CS 135, BSTs were used as *dictionaries*, with each node storing both a key and a value. Traditionally, a BST only stores a single item, and additional values can be added as *node augmentations* if required.

As with linked lists, we need a function to *create* a new BST.

```
// bst_create() creates a new BST
// effects: allocates memory: call bst_destroy

struct bst *bst_create(void) {
  struct bst *t = malloc(sizeof(struct bst));
  t->root = NULL;
  return t;
}
```


Before writing code to *insert* a new node, first we write a helper to create a new *leaf* node.

```
struct bstnode *new_leaf(int i) {
  struct bstnode *leaf = malloc(sizeof(struct bstnode));
  leaf->item = i;
  leaf->left = NULL;
  leaf->right = NULL;
  return leaf;
}
```

As with lists, we can write tree functions *recursively* or *iteratively*.

For the recursive version, we need to handle the special case that the tree is empty.

```
void bst_insert(int i, struct bst *t) {
  if (t->root) {
    insert_bstnode(i, t->root);
  } else {
    t->root = new_leaf(i);
  }
}
```


For the core function, we recurse on *nodes*.

```
void insert_bstnode(int i, struct bstnode *node) {
  if (i < node->item) {
    if (node->left) {
      insert_bstnode(i, node->left);
    } else {
      node->left = new_leaf(i);
    }
  } else if (i > node->item) {
    if (node->right) {
      insert_bstnode(i, node->right);
    } else {
      node->right = new_leaf(i);
    }
  } // else do nothing, as item already exists
}
```

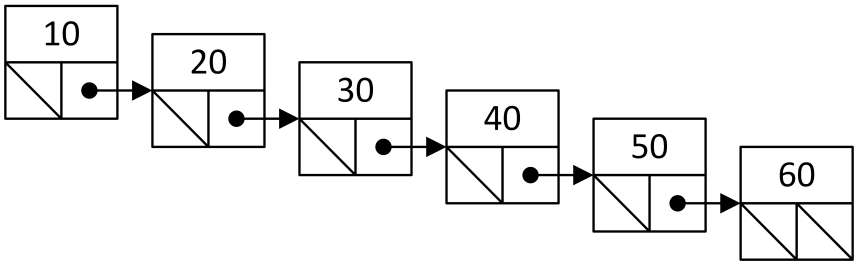

The iterative version is similar to the linked list approach.

```
void bst_insert(int i, struct bst *t) {
  struct bstnode *node = t->root;
  struct bstnode *parent = NULL;
  while (node) {
    if (node->item == i) return;
    parent = node;
    if (i < node->item) {
      node = node->left;
    } else {
      node = node->right;
    }
  }
  if (parent == NULL) {  // tree was empty
    t->root = new_leaf(i);
  } else if (i < parent->item) {
    parent->left = new_leaf(i);
  } else {
    parent->right = new_leaf(i);
  }
}
```

Trees and efficiency

What is the efficiency of `bst_insert`?

The *worst case* is when the tree is ***unbalanced***, and *every* node in the tree must be visited.



In this example, the running time of `bst_insert` is $O(n)$, where n is the number of nodes in the tree.

The running time of `bst_insert` is $O(h)$: it depends more on the *height* of the tree (h) than the *number of nodes* in the tree (n).

The definition of a ***balanced tree*** is a tree where the height (h) is $O(\log n)$.

Conversely, an **un**balanced tree is a tree with a height that is **not** $O(\log n)$. The height of an unbalanced tree is $O(n)$.

Using the `bst_insert` function we provided, inserting the nodes in *sorted order* creates an *unbalanced* tree.

With a **balanced** tree, the running time of standard tree functions (*e.g.,* `insert`, `remove`, `search`) are all $O(\log n)$.

With an **unbalanced** tree, the running time of each function is $O(h)$.

A ***self-balancing tree*** "re-arranges" the nodes to ensure that tree is always balanced.

With a good self-balancing implementation, all standard tree functions *preserve the balance of the tree* **and** have an $O(\log n)$ running time.

In CS 240 and CS 341 you will see *self-balancing trees*.

Self-balancing trees often use node augmentations to store extra information to aid the re-balancing.

Count node augmentation

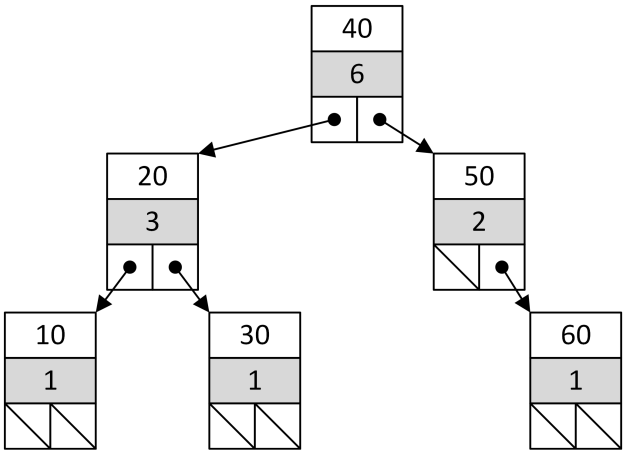
A popular tree **node augmentation** is to store in *each node* the **count** (number of nodes) in its subtree.

```
struct bstnode {
  int item;
  struct bstnode *left;
  struct bstnode *right;
  int count;                // *****NEW
};
```

This augmentation allows us to retrieve the number of nodes in the tree in $O(1)$ time.

It also allows us to implement a `select` function in $O(h)$ time. `select(k)` finds item with index `k` in the tree.

example: count node augmentation



The following code illustrates how to select item with index `k` in a BST with a `count` node augmentation.

```
int select_node(int k, struct bstnode *node) {
  assert(node && 0 <= k && k < node->count);
  int left_count = 0;
  if (node->left) left_count = node->left->count;
  if (k < left_count) return select_node(k, node->left);
  if (k == left_count) return node->item;
  return select_node(k - left_count - 1, node->right);
}

int bst_select(int k, struct bst *t) {
  return select_node(k, t->root);
}
```

`select(0, t)` finds the smallest item in the tree.

Dictionary ADT (revisited)

The dictionary ADT (also called a *map, associative array, key-value store or symbol table*), is a collection of **pairs** of ***keys*** and ***values***. Each *key* is unique and has a corresponding value, but more than one key may have the same value.

Typical dictionary ADT operations:

- **lookup:** for a given key, retrieve the corresponding value or "not found"
- **insert:** adds a new key/value pair (or replaces the value of an existing key)
- **remove:** *deletes* a key and its value

In the following example, we implement a Dictionary ADT using a BST data structure.

As in CS 135, we use `int` *keys* and `string` *values*.

```
// dictionary.h

struct dictionary;

struct dictionary *dict_create(void);

void dict_insert(int key, const char *val, struct dictionary *d);

const char *dict_lookup(int key, struct dictionary *d);

void dict_remove(int key, struct dictionary *d);

void dict_destroy(struct dictionary *d);
```


Using the same `bstnode` structure, we *augment* each node by adding an additional `value` field.

```
struct bstnode {
  int item;               // key
  char *value;            // additional value (augmentation)
  struct bstnode *left;
  struct bstnode *right;
};

struct dictionary {
  struct bstnode *root;
};

struct dictionary *dict_create(void) {
  struct dictionary *d = malloc(sizeof(struct dictionary));
  d->root = NULL;
  return d;
}
```

When inserting key/value pairs to the dictionary, we make a **copy** of the string passed by the client. When removing nodes, we also `free` the value.

If the client tries to insert a duplicate key, we replace the old value with the new value.

First, we will modify the `new_leaf` function to make a **copy** of the value provided by the client.

```
struct bstnode *new_leaf(int key, const char *val) {
  struct bstnode *leaf = malloc(sizeof(struct bstnode));
  leaf->item = key;
  leaf->value = my_strdup(val);     // make a copy
  leaf->left = NULL;
  leaf->right = NULL;
  return leaf;
}
```

And the insert is essentially the same:

```
void dict_insert(int key, const char *val, struct dictionary *d) {
  if (d->root) {
    insert_bstnode(key, val, d->root);
  } else {
    d->root = new_leaf(key, val);
  }
}
```



```
void insert_bstnode(int key, const char *val, struct bstnode *node)

  if (key == node->item) { // must replace the old value
    free(node->value);
    node->value = my_strdup(val);

  } else if (key < node->item) { // otherwise, it's the same
    if (node->left) {
      insert_bstnode(key, val, node->left);
    } else {
      node->left = new_leaf(key, val);
    }
  } else if (node->right) {
    insert_bstnode(key, val, node->right);
  } else {
    node->right = new_leaf(key, val);
  }
}
```

This implementation of the `lookup` operation returns `NULL` if unsuccessful.

```
const char *dict_lookup(int key, struct dictionary *d) {
  struct bstnode *node = d->root;
  while (node) {
    if (node->item == key) {
      return node->value;
    }
    if (key < node->item) {
      node = node->left;
    } else {
      node = node->right;
    }
  }
  return NULL;
}
```


There are several different ways of removing a node from a BST.

We implement `remove` with the following strategy:

- A) If the node with the key ("key node") is a leaf, we remove it.
- B) If one child of the key node is empty (`NULL`), the other child is "promoted" to replace the key node.
- C) Otherwise, we find the node with the *next largest* key ("next node") in the tree (*i.e.,* the smallest key in the right subtree). We replace the key/value of the key node with the key/value of the next node, and then remove the next node from the right subtree.


```
void dict_remove(int key, struct dictionary *d) {
  d->root = remove_bstnode(key, d->root);
}

struct bstnode *remove_bstnode(int key, struct bstnode *node) {
  // key did not exist:
  if (node == NULL) return NULL;

  // search for the node that contains the key
  if (key < node->item) {
    node->left = remove_bstnode(key, node->left);
  } else if (key > node->item) {
    node->right = remove_bstnode(key, node->right);
  } else if // continued on next page ...
            // (we have now found the key node)
```

If either child is `NULL`, the node is removed (`free`'d) and the other child is promoted.

```
  } else if (node->left == NULL) {
    struct bstnode *new_root = node->right;
    free(node->value);
    free(node);
    return new_root;
  } else if (node->right == NULL) {
    struct bstnode *new_root = node->left;
    free(node->value);
    free(node);
    return new_root;
  } else  // continued...
          // (neither child is NULL)
```

Otherwise, we replace the key/value at this node with next largest key/value, and then remove the next key from the right subtree...


```
  } else {
    // find next largest key and its parent
    struct bstnode *next = node->right;
    struct bstnode *parent_of_next = NULL;
    while (next->left) {
      parent_of_next = next;
      next = next->left;
    }
    // free old value & replace key/value of this node
    free(node->value);
    node->item = next->item;
    node->value = next->value;
    // remove next largest node
    if (parent_of_next) {
      parent_of_next->left = next->right;
    } else {
      node->right = next->right;
    }
    free(next);
  }
  return node;
}
```

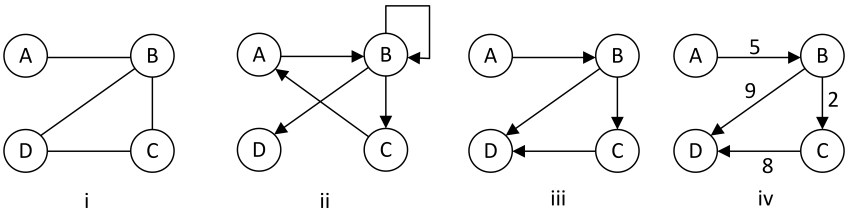

Finally, the recursive `destroy` operation `free`s the children and the (string) value before itself.

```
void free_bstnode(struct bstnode *node) {
  if (node) {
    free_bstnode(node->left);
    free_bstnode(node->right);
    free(node->value);
    free(node);
  }
}

void dict_destroy(struct dictionary *d) {
  free_bstnode(d->root);
  free(d);
}
```

Graphs

Linked lists and trees can be thought of as *"special cases"* of a ***graph*** data structure. Graphs are the only core data structure we are **not** working with in this course.



Graphs link **nodes** with **edges**. Graphs may be undirected (i) or directed (ii), allow cycles (ii) or be acyclic (iii), and have labeled edges (iv) or unlabeled edges (iii).

Goals of this Section

At the end of this section, you should be able to:

- use the new linked list and tree terminology introduced
- use linked lists and trees with a recursive or iterative approach
- use a cache and node augmentations to improve efficiency
- explain why an unbalanced tree can affect the efficiency of tree functions