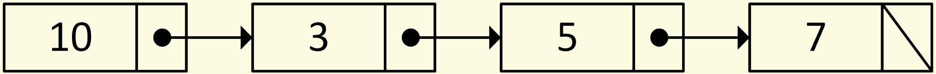
Module 10: Linked Data Structures

Readings: CP:AMA 17.5

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

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.

The *link* in the *last node* is a **sentinel value**.

In Racket we use `empty`, and in C we use a `NULL` pointer.

Linked lists

Recall from Racket:

```
;; A (listof Int) is one of:
;; * empty
;; * (cons Int (listof Int))
```

In C we will use a `struct` to store the pieces.

```
struct llnode {
  int data;              // "first"
  struct llnode *next; // "rest"
};
```

Then a `llnode *` that stores `NULL` is `empty`, and a `llnode` that points somewhere is a non-empty list.

To make a list longer, we need to create a new node. We will need to `malloc` more memory.

Exercise

Write a function `struct llnode *cons(int first, struct llnode *rest)` that creates a list that contains `first` before `rest`.

```
struct llnode *mynode = cons(10, cons(3, cons(5, cons(7, NULL))));
```

Using this function will cause a memory leak until we write code to `free` each `malloc`.

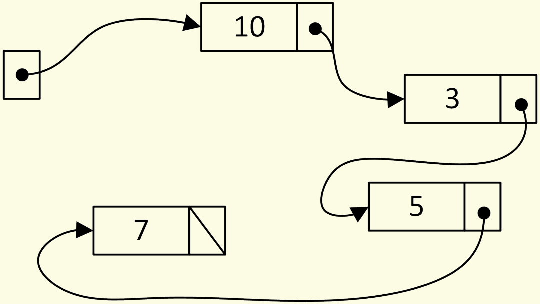
A linked list is usually represented as a pointer to the front.

```
struct llnode *mynode = cons(10, cons(3, cons(5, cons(7, NULL))));
```

Linked lists are not arranged sequentially in memory.

There is no way to “jump” to the *i*-th element.
We can only **traverse** them from the front.

We must `free` not only `myfirstnode`, but every `struct llnode` that we created. Recursively,




```
void llnode_destroy(struct llnode *node) {
  if (node == NULL) {
    // Base case is empty.
  } else {
    llnode_destroy(node->next);
    free(node);
  }
}
```

Linked lists: recursive traversal

```c
void llnode_destroy(struct llnode *node) {
  if (node == NULL) {
    // Base case is empty.
  } else {
    llnode_destroy(node->next);
    free(node);
  }
}
```

Exercise

Using `llnode_destroy` as a model, write a recursive function `void llnode_print(const struct llnode *node)` that prints the list, “Racket style”.

```c
struct llnode *mynode = cons(10, cons(3, cons(5, cons(7, NULL))));
llnode_print(mynode);
printf("\n");
llnode_destroy(mynode);
```

→ `(cons 10 (cons 3 (cons 5 (cons 7 empty))))`

Exercise

Write a recursive function `struct llnode *square_each(const struct llnode *node)` that returns a **new** linked list where each item has been squared.

```
struct llnode *mynode = cons(10, cons(3, cons(5, cons(7, NULL))));
struct llnode *squared = square_each(mynode);
llnode_print(mynode);
printf("\n");
llnode_print(squared);
printf("\n");
llnode_destroy(mynode);
llnode_destroy(squared);
```

→ `(cons 10 (cons 3 (cons 5 (cons 7 empty))))`

→ `(cons 100 (cons 9 (cons 25 (cons 49 empty))))`

Functional vs Imperative approach

In the functional programming paradigm, functions always generate **new** values rather than changing existing ones.

Consider a function that "squares" a list of numbers.

- In the *functional* paradigm, there is no **mutation**; it must generate a **new** list.
- in the *imperative* paradigm, it will likely to **mutate** an existing list.

Our `square_each` function is in the functional paradigm.

Functional vs Imperative approach

Exercise

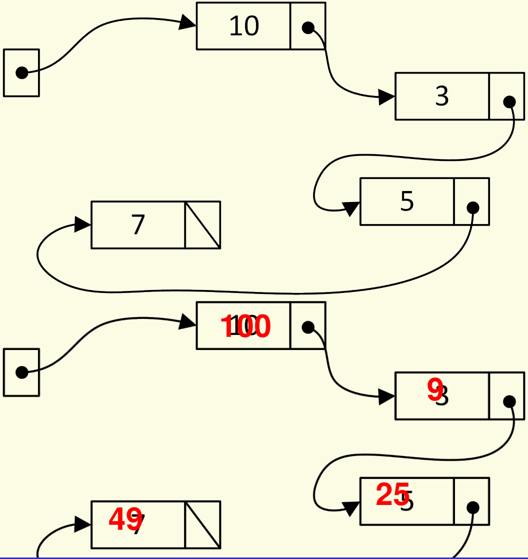
Write a **non**-recursive function

```
void square_each_mutate(struct llnode *node)
```

that **mutates** an existing linked list.

Use a `while` loop.

Consider: the parameter `node` points at the first node... but could point elsewhere.



In a similar way, we can write a non-recursive version of `llnode_destroy`:

```
void llnode_destroy(struct llnode *node) {
  while (node != NULL) {
    struct llnode *next = node->next;
    free(node);
    node = next;
  }
}
```

Exercise

Write a non-recursive version of `llnode_print`. **Make it end with a newline.**

```
  struct llnode *mynode = cons(10, cons(3, cons(5, cons(7, NULL))));
  llnode_print(mynode);
  llnode_destroy(mynode);
```

→ `(cons 10 (cons 3 (cons 5 (cons 7 empty))))`

Clear communication is particularly important here: is this function imperative or functional? Does it mutate, or does it create a new value?

In practice, most imperative list functions perform mutation. If the caller wants a new list (instead of mutating an existing one), they can first make a *copy* of the original list and then mutate the new copy.

Mixing paradigms

Problems may arise if we mix paradigms carelessly.

This is especially important in C, where there is no garbage collector.

Functional (Racket)	Imperative (C)
no mutation	mutation
garbage collector	no garbage collector
hidden pointers	explicit pointers

The following example highlights the potential problems.

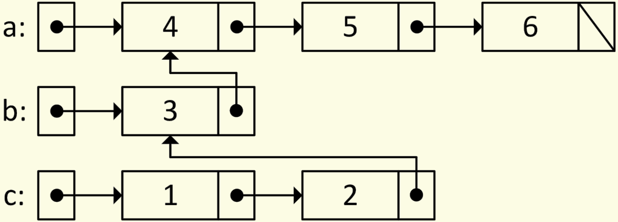
Node sharing

It is fine to **share nodes** in Racket:

```
(define a (list 4 5 6))
(define b (cons 3 a))
(define c (cons 1 (cons 2 b)))
```

It also seems fine in C:

```
struct llnode *a = cons(4, cons(5, cons(6, NULL)));
struct llnode *b = cons(3, a);
struct llnode *c = cons(1, cons(2, b));
```

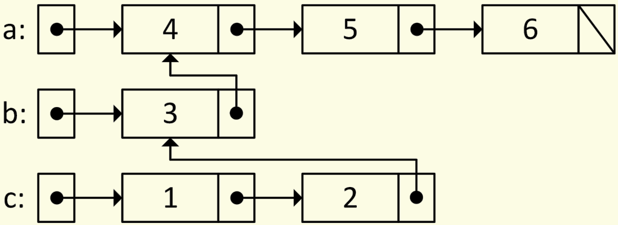


Caution:

If we forget to `llnode_destroy(c)` we get a **memory leak**.

If we `llnode_destroy(c)` but also `llnode_destroy(a)`, we get **heap-use-after-free**.

Node sharing



In Racket, lists can share nodes with no negative consequences; there is **no mutation**, and there is a **garbage collector**.

In an imperative language like C, this configuration is problematic.

- If we call a mutative function such as `square_each_mutate` on `a`, then some of the elements of `b` will unexpectedly change.
- If we explicitly `free` all of the memory for list `a`, then list `b` will become invalid.

Node sharing: wrappers

To avoid mixing paradigms, we use the following guidelines for linked lists in C:

- lists shall not **share nodes**.
- new nodes shall only be created to **insert** in an existing list, or to **create** a new list.

> We have already been following these guidelines. Notice:
>
> - Our recursive `square_each` **only** creates a new list.
>   It does not mutate or even link to the existing list.
> - Our non-recursive `square_each_mutate` **only** mutates the existing list.
>   It does not create any new nodes.

Node sharing: wrappers

To support our guidelines, we use a opaque **wrapper**:

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

Clients will interact only with `llist *` values; `llnode` values will be entirely hidden.

```
// list_create() Return a new, empty linked list.
// effects: allocates heap memory [caller must call list_destroy]
struct llist *list_create(void) {
  struct llist *llst = malloc(sizeof(struct llist));
  llst->front = NULL;
  return llst;
}
// list_destroy(ll) Clean up ll and its nodes.
void list_destroy(struct llist *ll) {
  llnode_destroy(ll->front);
  free(ll);
}
```

Node sharing: wrappers

```
struct llnode {
  int data;              // "first"
  struct llnode *next; // "rest"
};

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

A tiny bit of code supports what we wrote earlier:

```
// list_insert_front(ll, item) mutate ll
// so item comes before everything else.
void list_insert_front(struct llist *ll, int item) {
  ll->front = cons(item, ll->front);
}

// list_print(ll) Print ll, "Racket-style".
void list_print(const struct llist *ll) {
  llnode_print(ll->front);
}
```

Exercise

Write a function `void list_insert_back(struct llist *ll, int item)`

```
struct llist *ll = list_create();        → (cons 2 (cons 4 empty))
list_insert_back(ll, 2);
list_insert_back(ll, 4);
list_print(ll);
list_destroy(ll);
```

Hint

You will need special code to deal with the case when the front is the back.

Exercise

Also create a function `void list_insert_index(struct llist *ll, int item, int i)` that mutates `ll` to insert `item` so it has `i` items before it.

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 `NULL` at the end of the list.

Traversing a list

When using **recursion**, always recurse on a node (`llnode`) not the wrapper list itself (`llist`).

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

Write a corresponding wrapper function:

```
int list_length(struct llist *lst) {
  return length_nodes(lst->front, 0);
}
```

Node sharing: wrappers

Let us create a list:

```
struct llist *ll = list_create();
list_insert_front(ll, 7);
list_insert_front(ll, 5);
list_insert_front(ll, 3);
list_insert_front(ll, 10);
```

We seek to make a **copy** of a list. Try this:

```
// list_cp_bad(ll) Try to copy ll.
struct llist *list_cp_bad(struct llist *ll){
  struct llist *newlist =
    malloc(sizeof(struct llist));
  newlist->front = ll->front;
  return newlist;
}
```

It *seems* to work:

```
struct llist *ll_copy = list_cp_bad(ll);
list_print(ll);
list_print(ll_copy);
```

```
→ (cons 10 (cons 3 (cons 5 (cons 7 empty))))
→ (cons 10 (cons 3 (cons 5 (cons 7 empty))))
```

Exercise: Draw a memory diagram of this.

We see that these lists **share nodes**. This breaks our guidelines.

If we mutate nodes in `ll`, we also mutate nodes in `ll_copy`; they are the same nodes.

Exercise: Write a function `struct llist *list_cp(struct llist *ll)` that does not share nodes.
Hint: make it a wrapper around a recursive function on a `llnode *`.

Node sharing: wrappers

Some things are easier using recursion.
Then you should certainly practice writing them the “hard” way, using iteration!

Removing nodes

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

In C, we can implement a function that removes the first node.

```
void list_remove_front(struct llist *ll) {
  assert(ll);          // list is valid
  assert(ll->front); // list is not empty
  struct llnode *oldfront = ll->front;
  ll->front = ll->front->next;
  free(oldfront);
}
```

Exercise

Write a function `void list_remove_index(struct llist *ll, int i)` that removes item number `i` from `ll`. (0 is the first, 1 is the second, and so on.)

Hint

To remove a node, you must mutate the node that comes before it.

Example: removing by **value**

```
bool list_remove_value(struct llist *lst, int val) {
  if (lst->front == NULL) return false;
  if (lst->front->data == val) {
    list_remove_front(lst);
    return true;
  }
  struct llnode *prevnode = lst->front;
  while (prevnode->next && val != prevnode->next->data) {
    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;
}
```

Revisiting the wrapper approach

Throughout these slides we have used a *wrapper* strategy, where we wrap the link to the first node inside of another structure (`llist`).

Some advantages:

- cleaner function interfaces
- reduced need for double pointers
- reinforces the imperative paradigm
- less susceptible to misuse and list corruption

And disadvantages:

- slightly more awkward recursive implementations
- extra “special case” code around the first item

However, there is one more significant advantage of the wrapper approach: **additional information** can be stored in the list structure.

Augmentation

Imagine an application where we check the length of a linked list often.

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

But we can **augment** our ADT so it "caches" the length **in the wrapper structure**:

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

Exercise

Modify `list_create`, `list_insert_front`, `list_insert_back`, `list_remove_front`, and `list_remove_index` so they maintain this `length` field.
Then modify `list_length` so it runs in $O(1)$.

If we don't maintain this properly, we can end up with a structure that is **inconsistent**.

It could be wise to use an opaque ADT, to prevent users from erroneously writing something like `lst->length = 0;`

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 naive 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, but you must exercise caution.

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).

Stack ADT

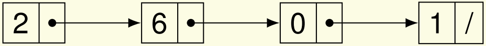
Exercise

Back in Module Module 6 we created a stack, but it had a fixed maximum depth.
Use a linked list to create a stack ADT that has no depth limit.
Implement the same methods we created before. Make the ADT opaque.

Queue ADT

A Stack ADT can be easily implemented using a dynamic array or linked list.

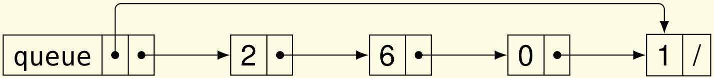
It is possible to implement a Queue ADT with a dynamic array, but it is a bit tricky. Queues are more often implemented with linked lists.



Problem: `list_insert_back` operation is $O(n)$.

Solution: **augment** the wrapper to maintain a pointer to the last element of the list, in addition to a pointer to the front of the list. Then `add_back` can be in $O(1)$.

```
struct queue {
  struct llnode *back;
  struct llnode *front;
};
```



queue.h

```c
// 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);
bool queue_is_empty(struct queue *q);
void queue_destroy(struct queue *q);
```

```c
struct queue {
  struct llnode *back;
  struct llnode *front;
};

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

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

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

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

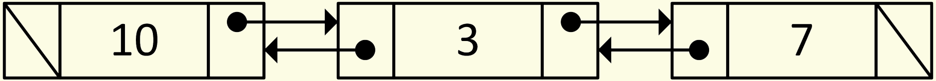
In a **node augmentation strategy**, each *node* is *augmented* to include additional information about the node or the structure.

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.

Node augmentation strategy

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 in a wrapper, 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.

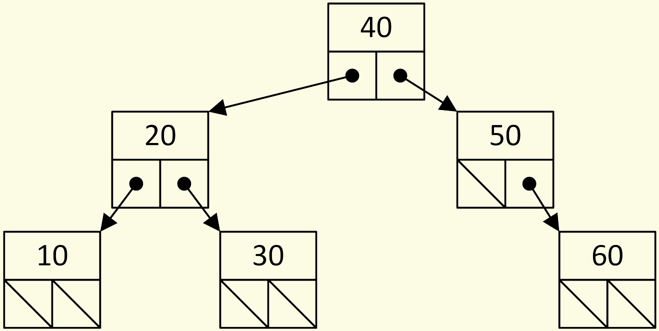
Exercise

Create a doubly-linked list ADT, which has operations **add-front**, **add-back**, **remove-front**, **remove-back**. Use a wrapper so all are in $O(1)$.

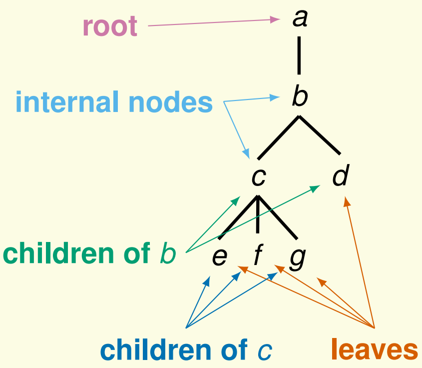
Trees

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

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

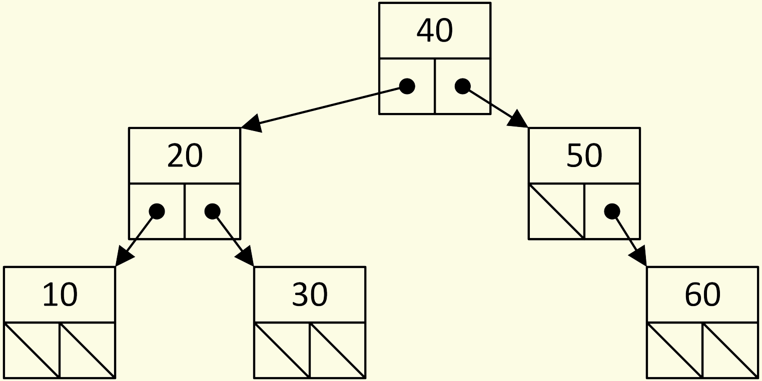


Tree terminology



- the **root** has no **parent**; all others have exactly 1.
- nodes can have multiple **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). Here it is 4.
- the height of an empty tree is zero.
- the number of nodes is known as the **node count**. Here it is 7.
- in a **binary tree**, each node has at most two children.

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** definition is similar to our linked list: a **wrapper**, and a recursive **node**.

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

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

Exercise

Write `bst_destroy(struct bst *b)`, and a recursive `bstnode_destroy(struct bstnode *n)` to clean these up.

Same code to create a wrapper:

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

```
void bstnode_destroy(struct bstnode *n) {
  if (n != NULL) {
    bstnode_destroy(n->left);
    bstnode_destroy(n->right);
    free(n);
  }
}

void bst_destroy(struct bst *b) {
  assert(b);
  bstnode_destroy(b->root);
  free(b);
}
```

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

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

Exercise

Write `bst_insert(struct bst *b, int item)`, and a recursive `bstnode_insert(struct bstnode *n, int item)`.
Maintain the **ordering property**: insert smaller items in the left, and larger in the right.

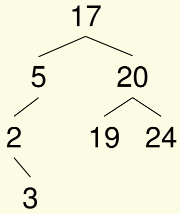
Insert solution

```
void bstnode_insert(struct bstnode *n, int item) {
  if (item < n->item) { // must insert to the left
    if (n->left == NULL) n->left = make_leaf(item);
    else bstnode_insert(n->left, item);
  } else if (item > n->item) { // must insert to the right
    if (n->right == NULL) n->right = make_leaf(item);
    else bstnode_insert(n->right, item);
  }
}

void bst_insert(struct bst *b, int item) {
  if (b->root == NULL) {
    b->root = make_leaf(item);
  } else {
    bstnode_insert(b->root, item);
  }
}
```

Similar code can traverse and print a tree:

```
void bstnode_print(struct bstnode *n, int depth)
    {
  if (n != NULL) {
    bstnode_print(n->right, depth + 1);
    // indent neatly:
    for (int i=0; i < depth; ++i) printf("    ");
    printf("%d\n", n->item);
    bstnode_print(n->left, depth + 1);
  }
}

void bst_print(struct bst *b) {
  if (b->root == NULL) printf("Empty tree\n");
  else bstnode_print(b->root, 0);
}
```

The if we insert $[17, 20, 5, 2, 3, 24, 19]$:



Which prints like this (tilt your head ↶):

```
        24
    20
        19
17
    5
            3
        2
```

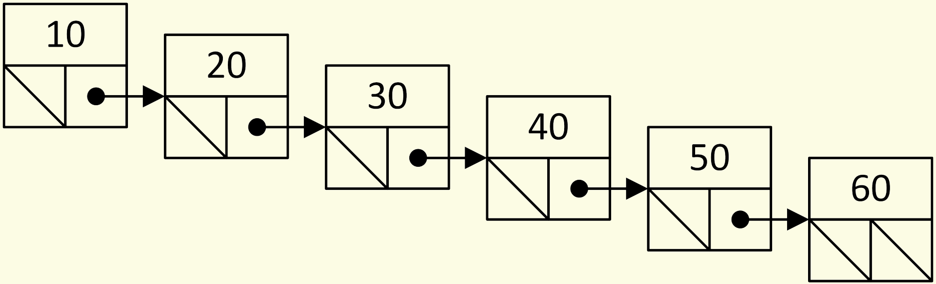
BST printing

We could also make iterative version, but it's harder:

```
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);
```

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.

Trees and efficiency

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).

If a tree is **balanced**, its height will be $O(\log n)$.

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.

Trees and efficiency

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 $O(n)$.

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.

CS 234, CS 240, and CS 341 discuss *self-balancing trees*.
Self-balancing trees often use node augmentations to store extra information to aid the re-balancing.

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

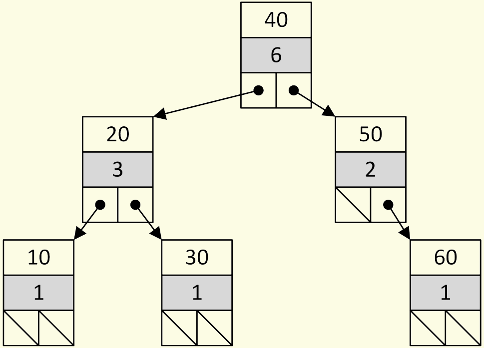
```
struct bstnode {
  int item;
  struct bstnode *left;
  struct bstnode *right;
  int count;              // *****NEW
};
```

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.

Count node augmentation

example: count node augmentation



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);
}
```

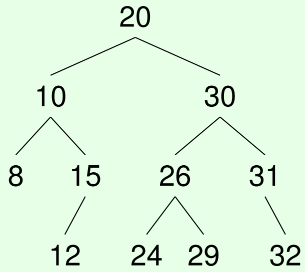
`select(0, t)` finds the smallest item in the tree.

For some types of trees, it is possible to use an **array** to store a tree.

- the root is stored in element 0
- for the node in element i,
  - its `left` is stored int $2i + 1$
  - its `right` is stored int $2i + 2$
  - its `parent` is stored int $\frac{i-1}{2}$
- a special *sentinel value* can be used to indicate an empty node
- a tree of height h requires an array of length $2^h - 1$
  (a dynamic array can be `realloc`'d as the tree height grows)

Exercise

Create an array to represent the following tree. Write `NULL` as a sentinel.



Exercise

Write function that takes a pointer to the root, and the maximum number of values in the tree. It returns the sum of all the node values.
For example, if `tree` is the array from the previous slide,
`sum_tree(tree, 15)` $\Longrightarrow$ `20 + 10 + 30 + 8 + 15 + 26 + 31 + 12 + 24 + 29 + 32` $\Longrightarrow$ `237`

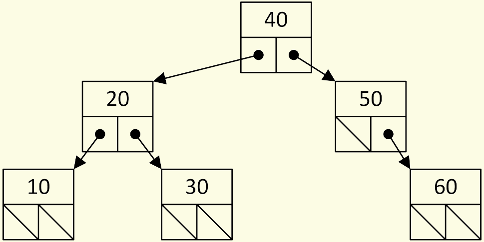
Array-based trees

example: array-based tree representation

`left: 2i+1`

`right: 2i+2`

40	20	50	10	30	-	60



Array-based trees

Array-based trees are often used to implement "complete trees", where there are no *empty* nodes, and every level of the tree is filled (except the bottom).

The *heap* data structure (not the section of memory) is often implemented as a complete tree in an array.

For *self-balancing* trees, the self-balancing (e.g. rotations) is often more awkward in the array notation. However, arrays work well with *lazy* rebalancing, where a rebalancing occurs infrequently (i.e. when a large inbalance is detected). The tree can be rebalanced in $O(n)$ time, typically achieving *amortized* $O(\log n)$ operations.

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 wrapper structures and node augmentations to improve efficiency
- explain why an unbalanced tree can affect the efficiency of tree functions