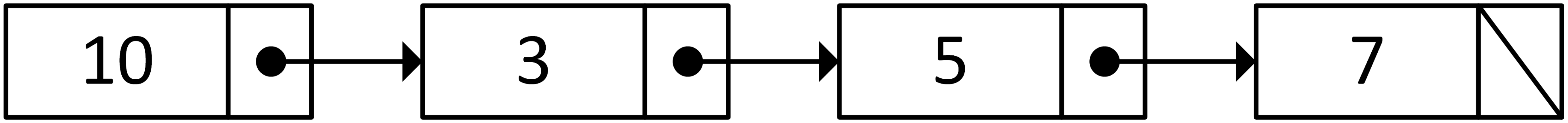
Linked Data Structures

Readings: CP:AMA 17.5

Linked lists

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

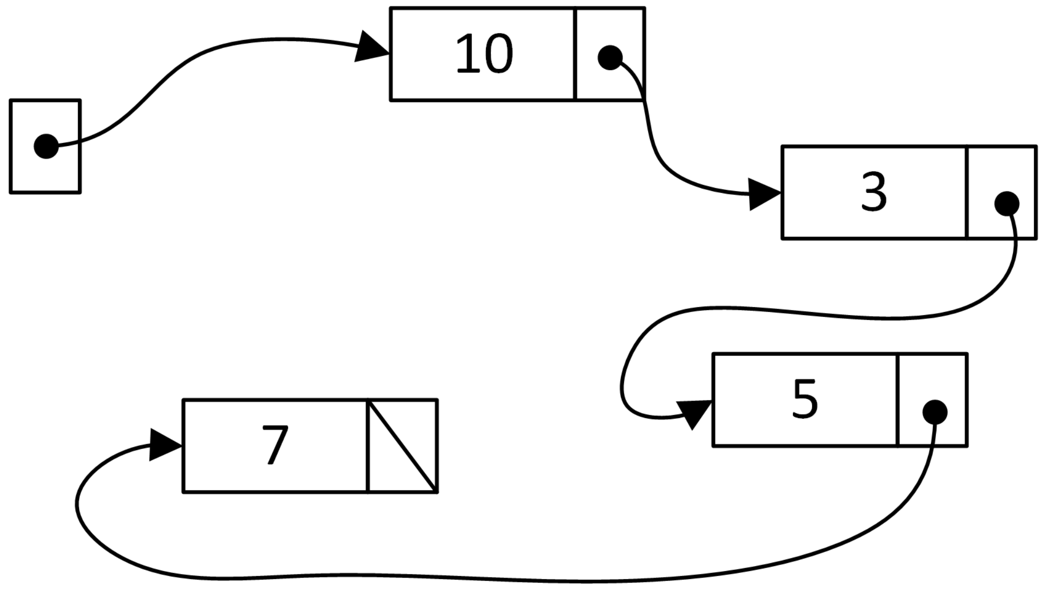


Each ***node*** contains an ***item*** and a **link** (*pointer*) to the ***next*** node in the list.

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

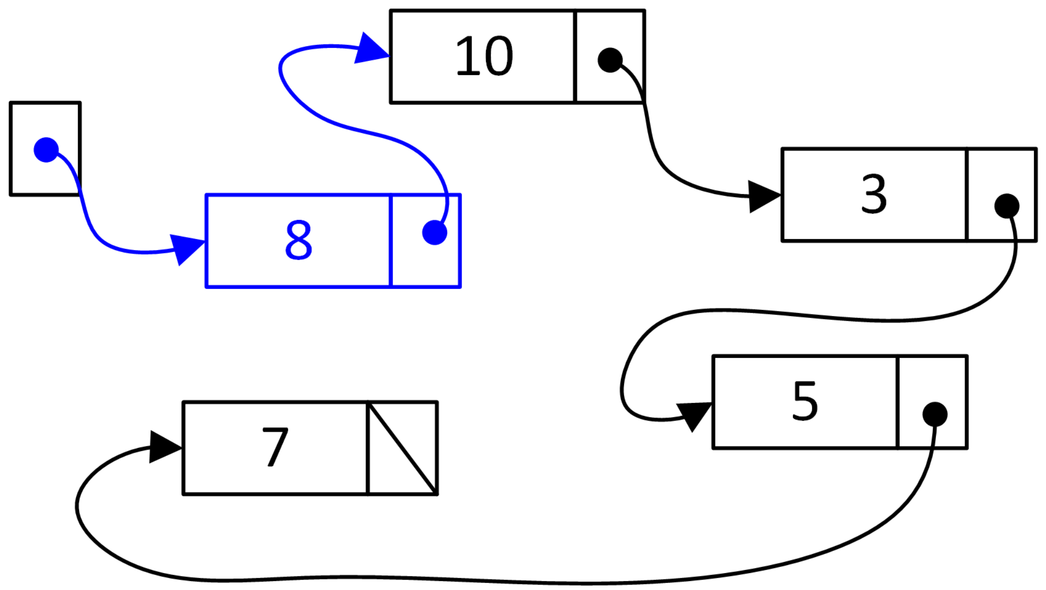
In Racket we use `empty`, and in C we use a `NULL` pointer.

Linked lists 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*).

Functional *vs.* Imperative approach

In Section 04, we discussed some of the differences between the **functional** and **imperative** programming paradigms.

The core concept of a linked list data structure is independent of any single paradigm.

However, the approach used to **implement** linked list functions are often very different.

Programming with linked lists further illustrates the differences between the two paradigms.

Dynamic memory in Racket

Dynamic memory in Racket is mostly *"hidden"*.

The `cons` function dynamically creates a **new** linked list **node**.

In other words, inside of every `cons` is a hidden `malloc`.

Structure constructors also use dynamic memory

(*e.g.*, `make-posn` or simply `posn` in full Racket).

`list` and quote list notation `'(1 2 3)` *implicitly* use `cons`.

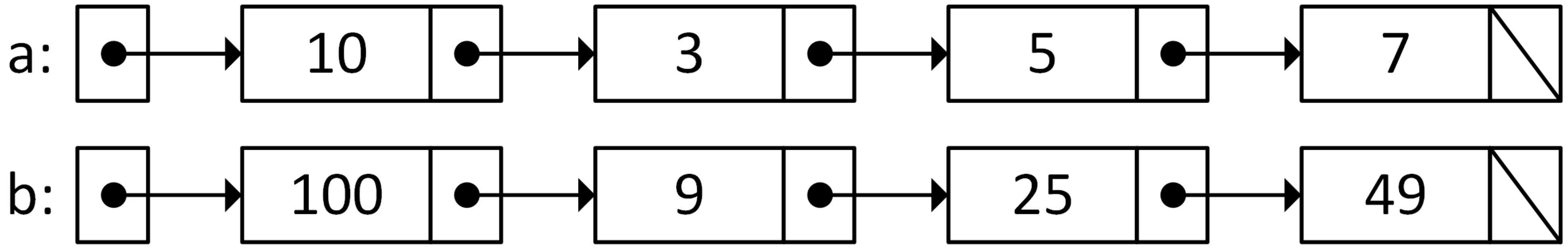
In the functional programming paradigm, functions always produce **new** values rather than changing existing ones.

Consider a function that “squares” a list of numbers.

- In the *functional* paradigm, the function **must** produce a **new** list, because there is no *mutation*.
- in the *imperative* paradigm, the function is more likely to **mutate** an existing list instead of producing a new list.

`sqr-list` uses `cons` to `const`ruct a **new list**:

```
(define (sqr-list lst)
  (cond [(empty? lst) empty]
        [else (cons (sqr (first lst))
                    (sqr-list (rest lst)))]))

(define a '(10 3 5 7))
(define b (sqr-list a))
```



Of course, in an imperative language (*e.g.,* C) it is *also* possible to write a "square list" function that follows the functional paradigm and generates a new list.

This is another example of why clear communication (purposes and contracts) is so important.

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 naïvely use the functional paradigm in an imperative environment without considering the consequences.

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

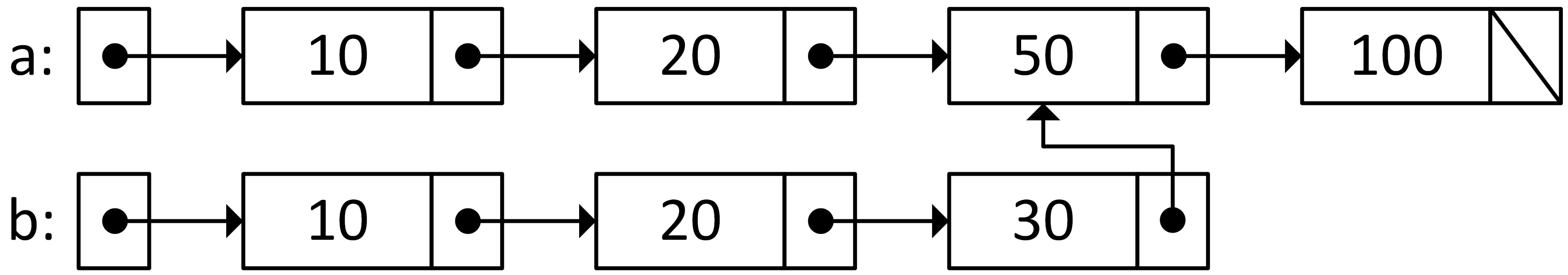
Functional (Racket)	Imperative (C)
no mutation	mutation
garbage collector	no garbage collector
hidden pointers	explicit pointers

The following example highlights the potential problems.

Consider an `insert` function (used in `insertion sort`).

```
;; (insert n slon) inserts n into a sorted list of numbers

(define (insert n slon)
  (cond [(empty? slon) (cons n empty)]
        [(<= n (first slon)) (cons n slon)]
        [else (cons (first slon) (insert n (rest slon)))]))

(define a '(10 20 50 100))
(define b (insert 30 a))
```

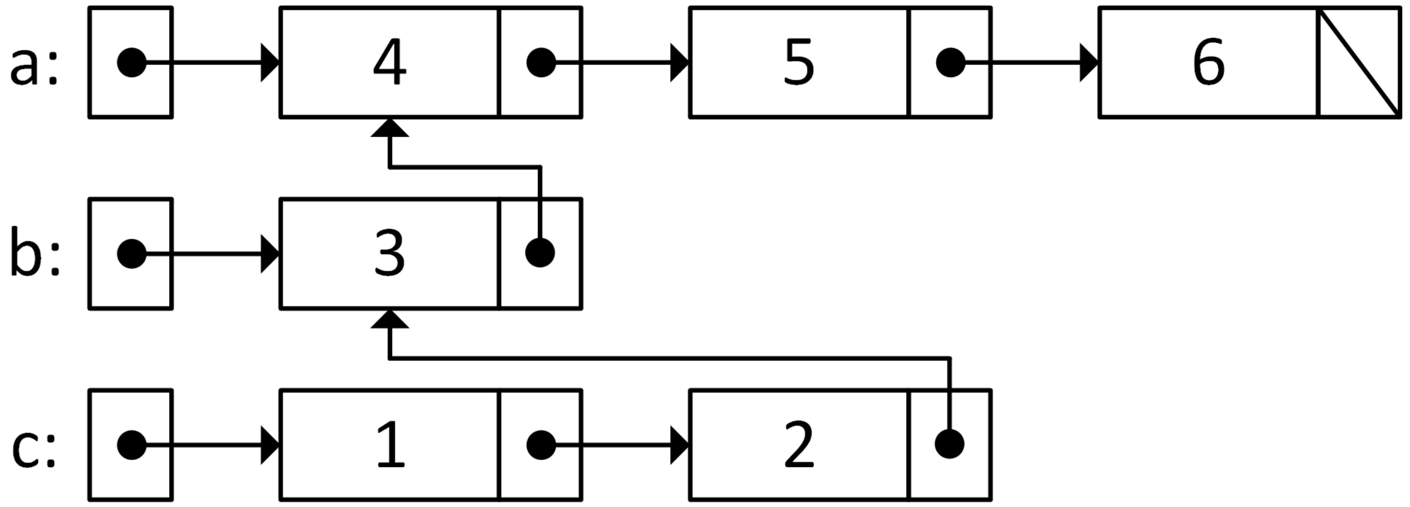
What will the memory diagram look like for a and b?

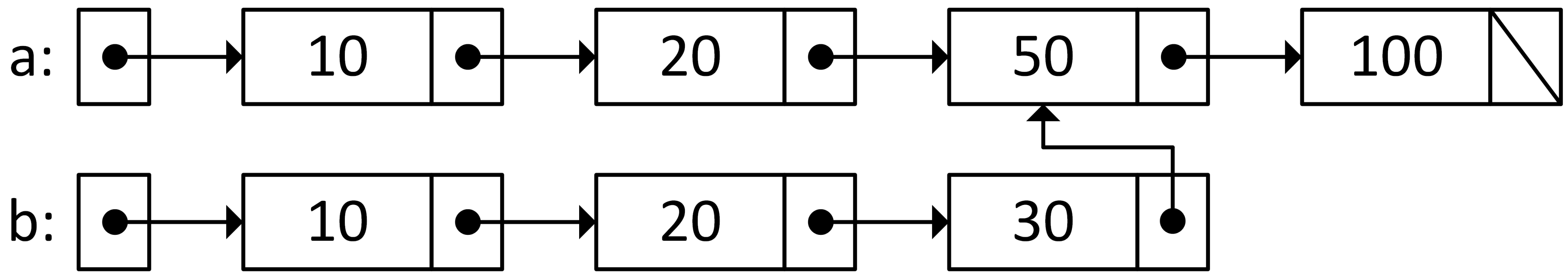


The lists **share the last two nodes**.

example: more node sharing in Racket

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





In Racket, lists can share nodes with no negative consequences.

It is *transparent* because there is **no mutation**, and there is a **garbage collector**.

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

- If we apply a mutative function such as “square list” 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.

To avoid mixing paradigms, we will use the following *guidelines* when implementing linked lists in C:

- lists will not **share nodes**
- new nodes will only be created (`malloc`'d) when necessary (inserting nodes and creating new lists).

In Racket, lists generated with `cons` are immutable.

There is a special `mcons` function to generate a mutable list.

This is one of the significant differences between the Racket language and the Scheme language.

In the Scheme language, lists generated with `cons` are mutable.

Linked lists in C

We declare a *linked list node* (`llnode`) that stores an "item" and a link (pointer) to the next node. For the *last node*, `next` is `NULL`.

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

A C structure can contain a *pointer* to its own structure type. This is the first **recursive data structure** we have seen in C.

There is no "official" way of implementing a linked list in C. The CP:AMA textbook and other sources use slightly different conventions.

In this Section we use a ***wrapper strategy***, where we wrap the link to the first node of a list inside of another structure (`llist`).

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

This wrapper strategy makes some of the following code more straightforward.

It also makes it easier to avoid having lists share nodes (mixing paradigms).

In Appendix A.8, we present examples that do not use a wrapper and briefly discuss the advantages and disadvantages.

To dynamically create a list, we define a `list_create` function.

```
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();
  // ...
}
```

We need to add items to our linked list.

The following code creates a **new** node, and inserts it at the **front** of the list.

```
void add_front(int i, struct llist *lst) {
  struct llnode *node = malloc(sizeof(struct llnode));
  node->item = i;
  node->next = lst->front;
  lst->front = node;
}
```

example: building a linked list

The following code builds a linked list by reading in integers from the input.

```c
int main(void) {

  struct llist *lst = list_create();

  while(1) {
    int i;
    if (scanf("%d", &i) != 1) break;
    add_front(i, lst);
  }
  // ...
}
```

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

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

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

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

You can write a corresponding wrapper function:

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

or call the recursive function directly on the `front` of the list.

```
int len = 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 wrapper.

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;
  while (curnode) {
    struct llnode *nextnode = curnode->next;
    free(curnode);
    curnode = nextnode;
  }
  free(lst);
}
```

For more advanced list traversal functions, the technique of maintaining more than one node pointer is often necessary.

It may take some practice and diagrams to master this technique.

For extra practice, consider this slightly different implementation:

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

Previously, we used the “square list” function to illustrate the differences between the functional and imperative paradigms.

```
// list_sqr(lst) squares each item in lst
// effects: modifies lst

void list_sqr(struct llist *lst) {
  struct llnode *node = lst->front;
  while (node) {
    node->item *= node->item;
    node = node->next;
  }
}
```

But what if we do want a **new** list that is squared instead of mutating an existing one?

One solution is to provide a `list_dup` function, that makes a *duplicate* of an existing list.

The recursive function is the most straightforward.

```
void dup_nodes(struct llnode *oldnode,
               struct llist *newlist) {
  if (oldnode) {
    dup_nodes(oldnode->next, newlist);
    add_front(oldnode->item, newlist);
  }
}

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

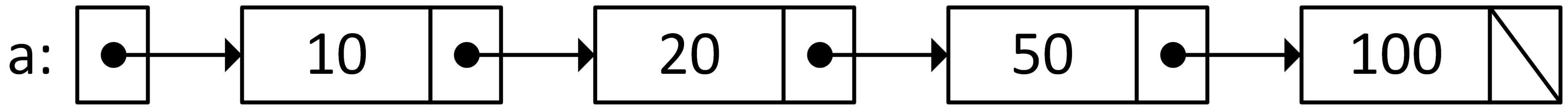
The iterative solution is more complicated:

```
struct llist *list_dup(struct llist *oldlist) {
  struct llist *newlist = list_create();
  struct llnode *oldnode = oldlist->front;
  struct llnode *prevnode = NULL;
  while (oldnode) {
    struct llnode *newnode = malloc(sizeof(struct llnode));
    newnode->item = oldnode->item;
    newnode->next = NULL;
    if (prevnode) {
      prevnode->next = newnode;
    } else {
      newlist->front = newnode;
    }
    prevnode = newnode;
    oldnode = oldnode->next;
  }
  return newlist;
}
```

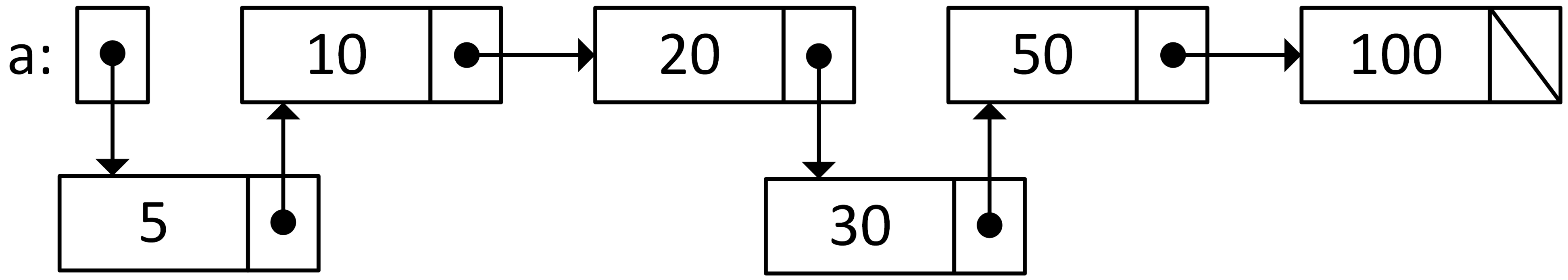
Imperative insert

Earlier, we saw how the Racket (functional) implementation of *insert* (into a sorted list) would be problematic in C.

For an `insert` function in C, we expect the following behaviour:



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



```
// insert(i, slst) inserts i into sorted list slst
// 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;
    }
    struct llnode *newnode = malloc(sizeof(struct llnode));
    newnode->item = i;
    newnode->next = prevnode->next;
    prevnode->next = newnode;
  }
}
```

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.

```
int remove_front(struct llist *lst) {
  assert(lst->front);
  int retval = lst->front->item;
  struct llnode *backup = lst->front;
  lst->front = lst->front->next;
  free(backup);
  return retval;
}
```

Instead of `return`ing nothing (`void`), it is more useful to `return` the value of the item being removed.

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

```
// remove_item(i, lst) removes the first occurrence of i in lst
//   return value indicates 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 *backup = prevnode->next;
  prevnode->next = prevnode->next->next;
  free(backup);
  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 of the advantages of this strategy are:

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

The disadvantages of the wrapper approach include:

- 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.

See Appendix A.8 for more details.

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 wrapper 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, 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).

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 $O(1)$.

Maintaining a `back` pointer is a popular modification to a traditional linked list, and another reason to use a wrapper.

```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);
```

```
// 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;
}
```

```
void queue_add_back(int i, struct queue *q) {
  struct llnode *node = malloc(sizeof(struct llnode));
  node->item = i;
  node->next = 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 *backup = q->front;
  q->front = q->front->next;
  free(backup);
  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);
}
```

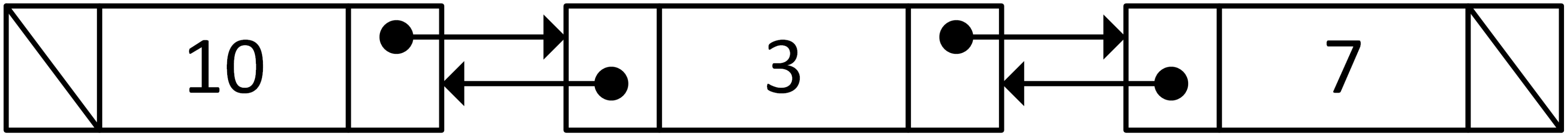
Node augmentation strategy

In an **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 in a wrapper, a doubly linked list can add or remove from the front **and back** in $O(1)$ time.

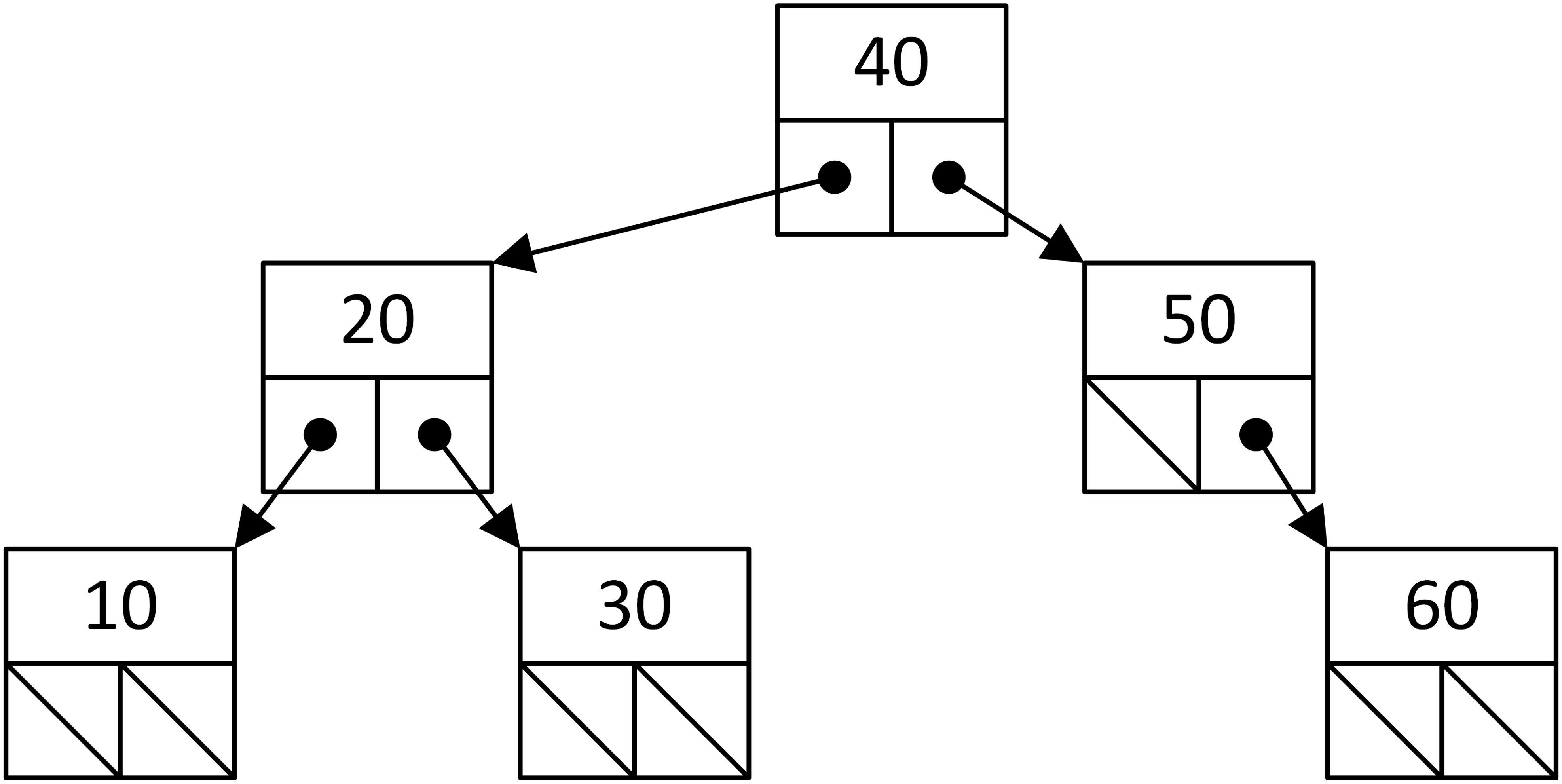


Many programming environments provide a Double-Ended Queue (dequeue or deque) ADT, which can be used as a Stack or a Queue ADT.

Trees

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

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

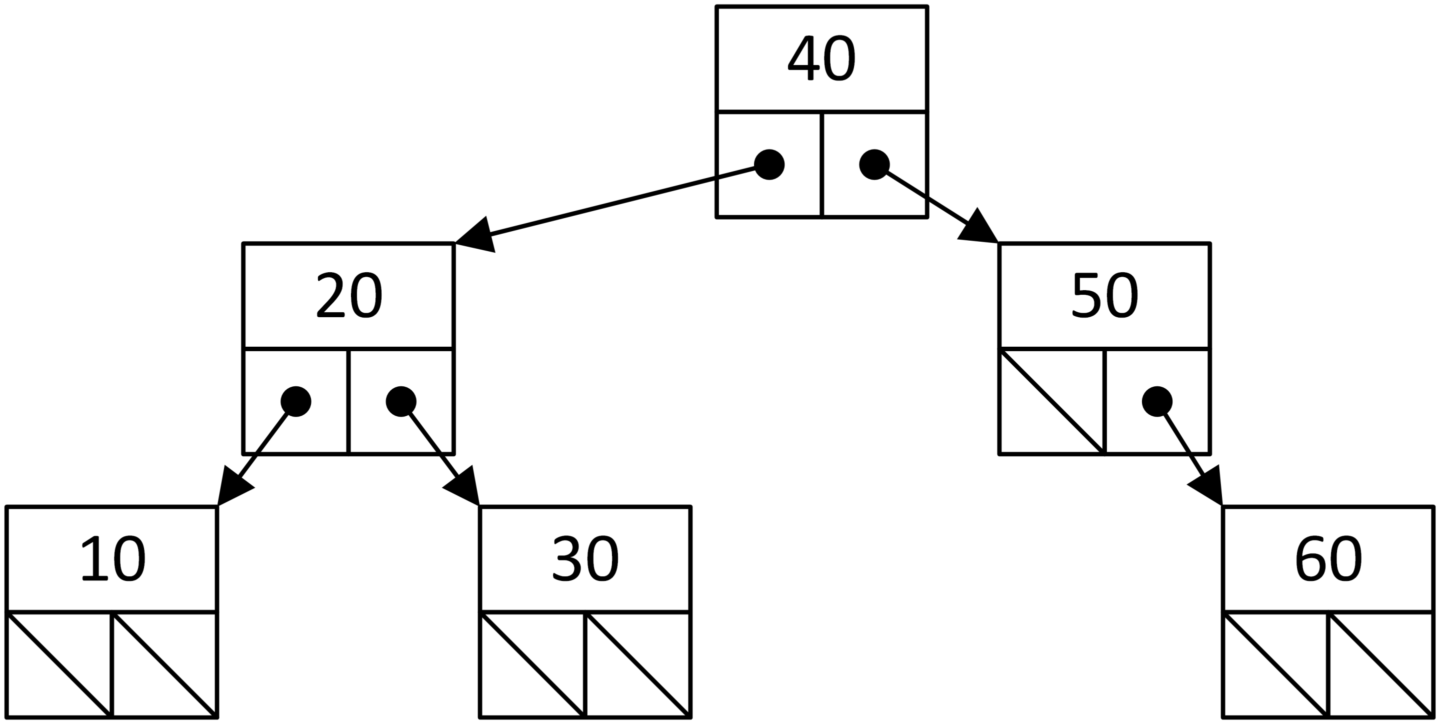


Tree terminology

- the **root node** has no **parent**
- all other nodes have exactly one parent
- 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

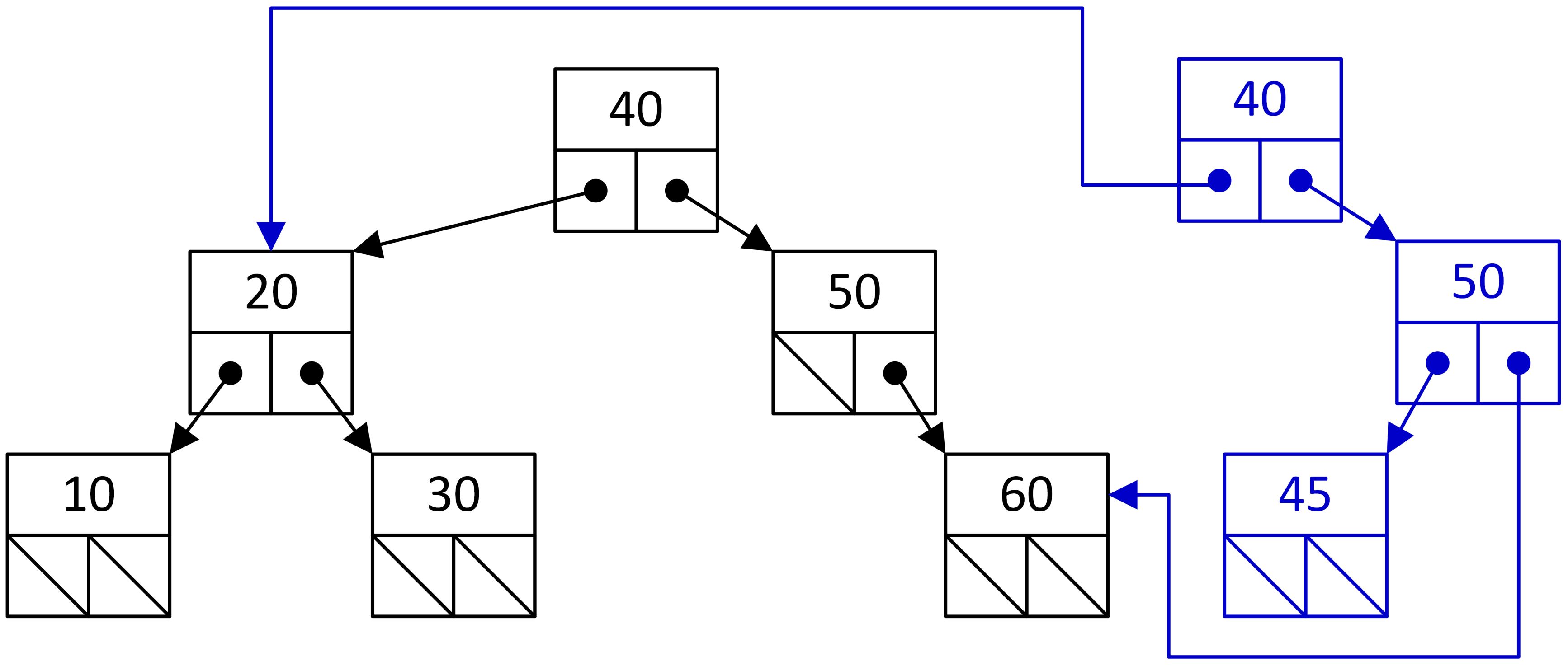
Binary Search Trees (BSTs)

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.



Mixing paradigms

As with linked lists, we have to be careful not to mix functional and imperative paradigms, especially when adding nodes. The following example visualizes what Racket produces when a node (45) is added to the BST illustrated earlier.



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 will 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*.

We need to **recurse** on *nodes*. This code emulates a functional approach, but is careful to only allocate one new (leaf) node.

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

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

The iterative version is similar to the linked list approach.

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

example: building a BST

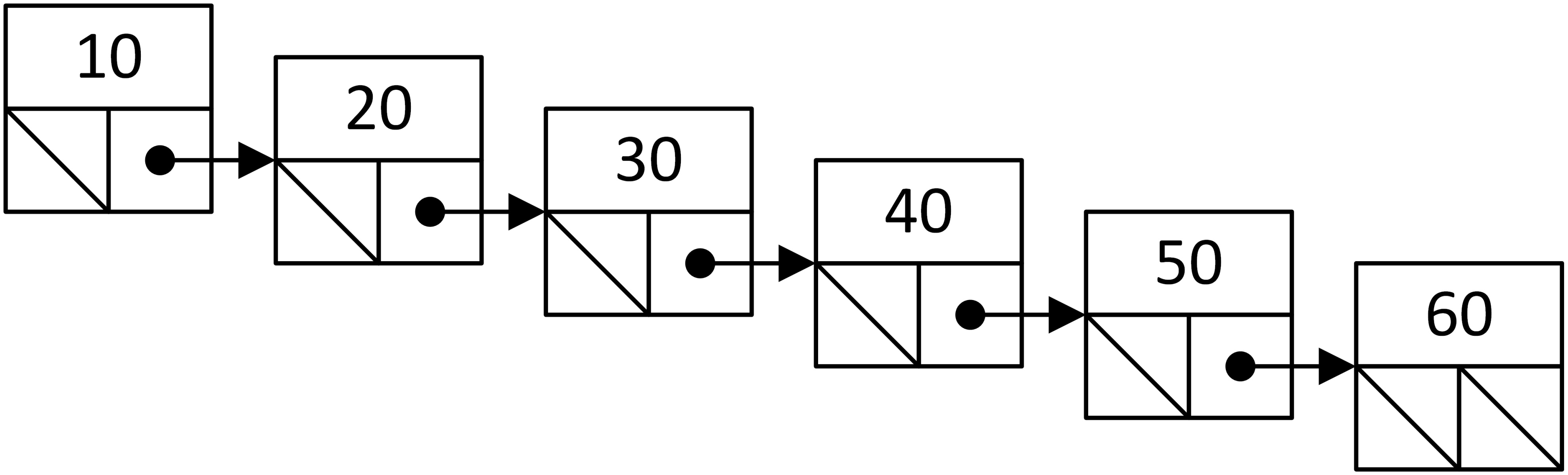
The following code builds a BST by reading in integers from the input.

```
int main(void) {

  struct bst *t = bst_create();

  while(1) {
    int i;
    if (scanf("%d", &i) != 1) break;
    bst_insert(i, t);
  }
  // ...
}
```

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 *size* of 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.

Size node augmentation

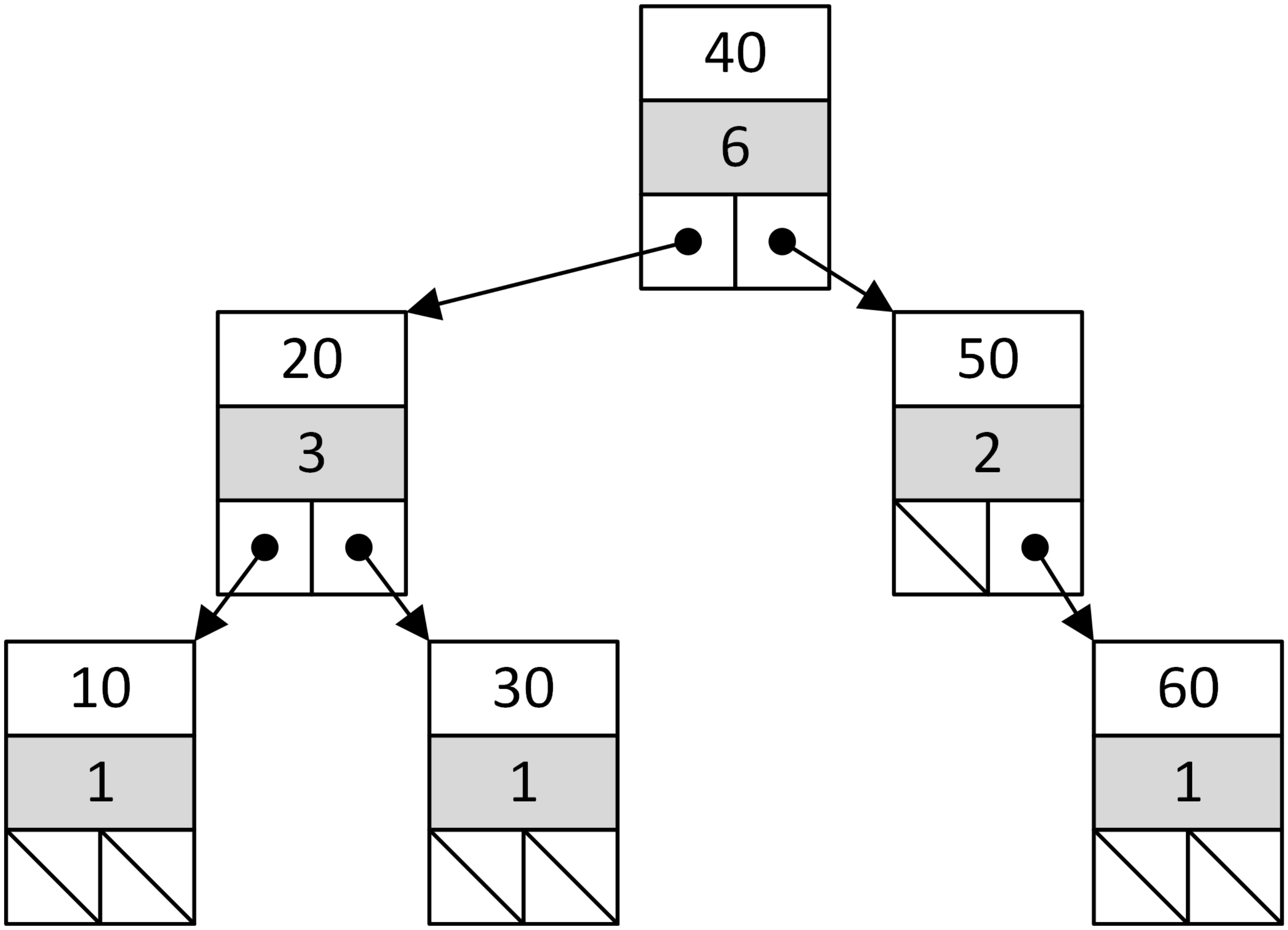
A popular tree **node augmentation** is to store in *each node* the **size** of its subtree.

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

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

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

example: size node augmentation



The following code illustrates how to select the `k`-th item in a BST with a `size` node augmentation.

```
int select_node(int k, struct bstnode *node) {
  assert(node && 0 <= k && k < node->size);
  int left_size = 0;
  if (node->left) left_size = node->left->size;
  if (k < left_size) return select_node(k, node->left);
  if (k == left_size) return node->item;
  return select_node(k - left_size - 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.

Array-based trees

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

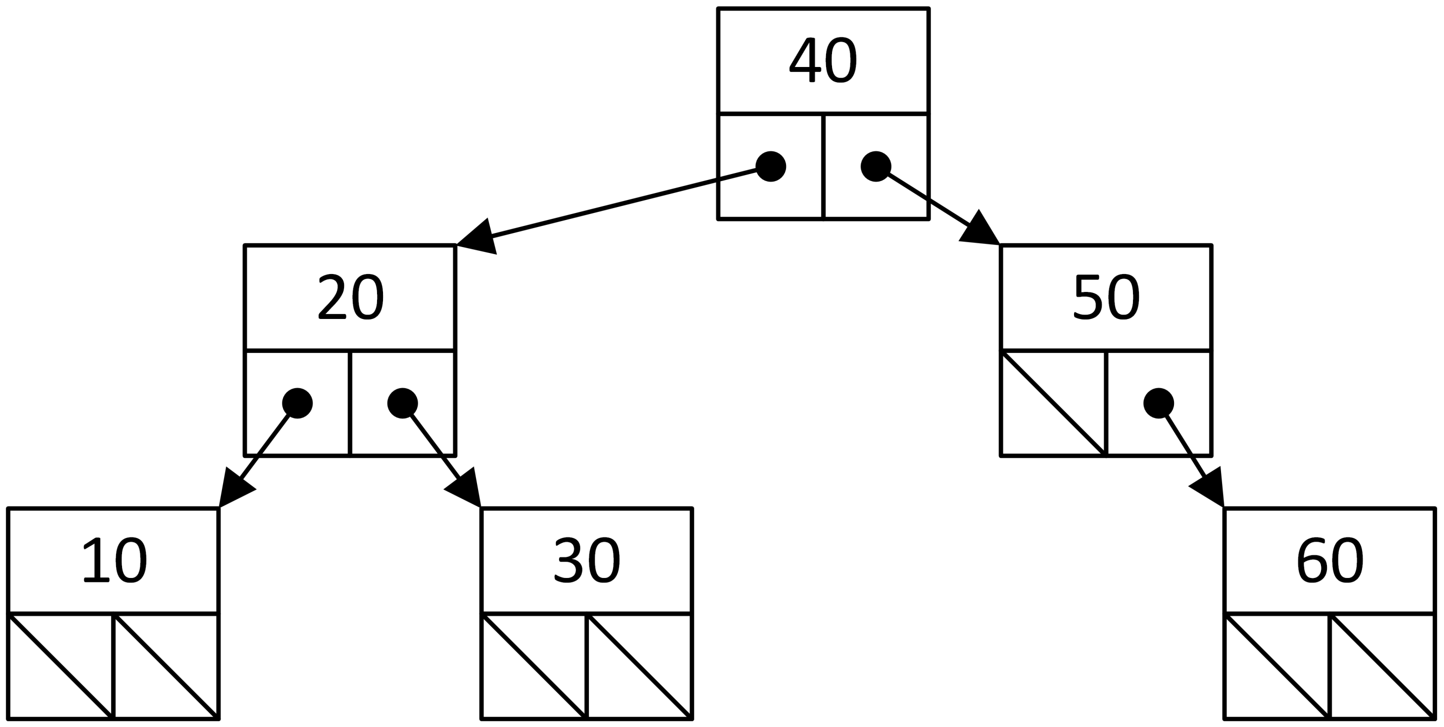
- the root is stored at `a[0]`
- for the node at `a[i]`, its `left` is stored at `a[2*i+1]`
- its `right` is stored at `a[2*i+2]`
- its `parent` is stored at `a[(i-1)/2]`
- a special *sentinel value* can be used to indicate an empty node
- a tree of height h requires an array of size $2^h - 1$

  (a dynamic array can be `realloc`'d as the tree height grows)

example: array-based tree representation

```
left:  2i+1
right: 2i+2
```

40	20	50	10	30	-	60



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.

Dictionary ADT (revisited)

The dictionary ADT (also called a *map, associative array, 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 will 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
  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.

The following *recursive* implementation of the `insert` operation is nearly identical to our previous `bst_insert`. The differences are noted with comments.

```
struct bstnode *insert_bstnode(int key, const char *val,
                               struct bstnode *node) {
  if (node == NULL) {
    node = malloc(sizeof(struct bstnode));
    node->item = key;
    node->value = my_strdup(val);         // make copy
    node->left = NULL;
    node->right = NULL;
  } else if (key < node->item) {
    node->left = insert_bstnode(key, val, node->left);
  } else if (key > node->item) {
    node->right = insert_bstnode(key, val, node->right);
  } else { // key == node->item: must replace the old value
    free(node->value);
    node->value = my_strdup(val);
  }
  return node;
}

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

This implementation of the `lookup` operation will return `NULL` if unsuccessful.

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

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

We implement `remove` with the following strategy:

- If the node with the key (“key node”) is a leaf, we remove it.
- If one child of the key node is empty (`NULL`), the other child is “promoted” to replace the key node.
- 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.

```c
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 *backup = node->right;
  free(node->value);
  free(node);
  return backup;
} else if (node->right == NULL) {
  struct bstnode *backup = node->left;
  free(node->value);
  free(node);
  return backup;
} 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.

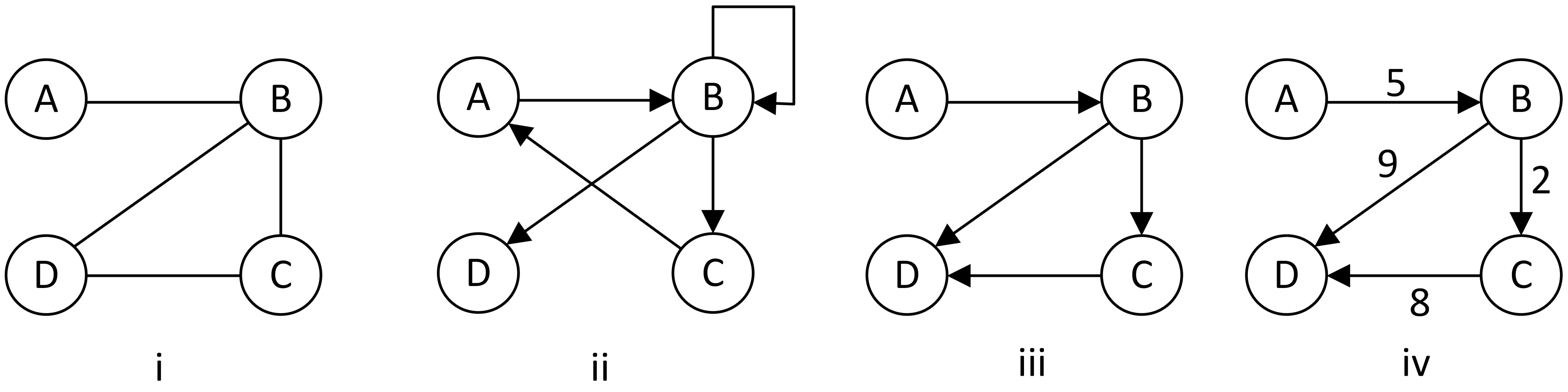
```
    } else {
      // find the next largest key
      struct bstnode *next = node->right;
      while (next->left) {
        next = next->left;
      }
      // remove the old value
      free(node->value);
      // replace the key/value of this node
      node->item = next->item;
      node->value = my_strdup(next->value);
      // remove the next largest key
      node->right = remove_bstnode(next->item, node->right);
    }
    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 wrapper structures and node augmentations to improve efficiency
- explain why an unbalanced tree can affect the efficiency of tree functions