CSC215 Lecture

Data Structures in C

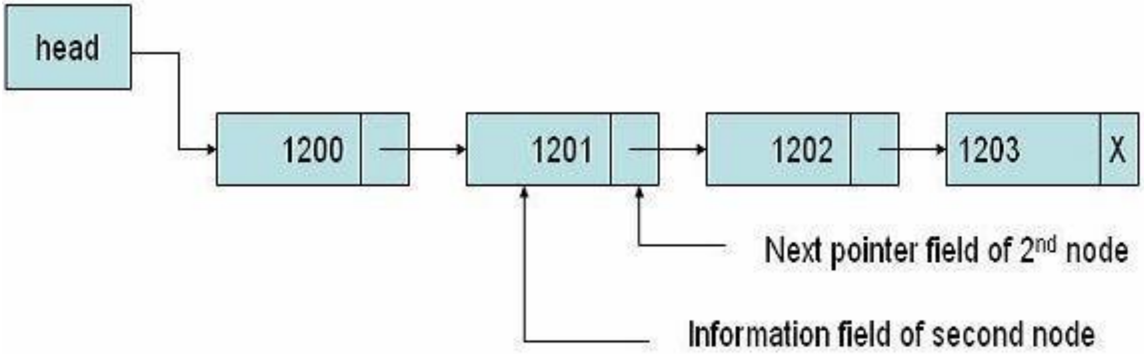
Outline

- Linked Lists
- Binary Trees
- Stacks
- Queues
- Hash Tables

Linked Lists

- Linked List: A dynamic data structure that consists of a sequence of nodes
  - each element contains a link or more to the next node(s) in the sequence
  - Linked lists can be singly or doubly linked, linear or circular.
- Every node has a payload and a link to the next node in the list
- The start (head) of the list is maintained in a separate variable
- End of the list is indicated by NULL (sentinel).
- Example:

```
struct Node{
   void* data;
   struct Node* next;
};
struct LinkedList {
   struct Node* head;
};
```



Linked Lists: Operations

```
typedef struct Node Node;

Node* new_node(void*);

typedef struct LinkedList LinkedList;

LinkedList* new_linked_list();

void insert_at_front(LinkedList*,void*);

void insert_at_back(LinkedList*,void*);

void* remove_from_front(LinkedList*);

void* remove_from_back(LinkedList*);

int size(LinkedList*);

int is_empty(LinkedList*);
```

```
struct Node{
  void* data;   Node* next;
};

Node* new_node(void* data){
 Node* n=(Node*)
          calloc(1,sizeof(Node));
 n->data = data;
 return n;
}

struct LinkedList {
 Node* head;
};

LinkedList* new_linked_list(){
 LinkedList* ll=(LinkedList*)
    calloc(1,sizeof(LinkedList));
  return ll;
}
```

Linked Lists: Operations

- Iterating:
  - `for (p=head; p!=NULL; p=p->next) /* do something */`
  - `for (p=head; p->next !=NULL; p=p->next) /* do something */`
  - `for (p=head; p->next->next !=NULL; p=p->next) /* do something */`

```
int size(LinkedList* ll){
  int result = 0;
  Node* p = ll->head;
  while (p){
    p=p->next; result++;
  }
  return result;
}
```

```
int is_empty(LinkedList* ll) {
  return !ll->head;
}
```

Linked Lists: Operations - insert

```
void insert_at_front(LinkedList* ll, void* data){
  Node* n = new_node(data);
  if (!n) return;
  n->next = ll->head;
  ll->head = n;
}
```

```
void insert_at_back(LinkedList* ll, void* data){
  Node* n = new_node(data);
  if (!n) return;
  Node* p = ll->head;
  if (!p) ll->head = n;
  else {
    while (p->next) p=p->next;
    p->next = n;
  }
}
```

Linked Lists: Operations - insert

```
void insert_after_nth(LinkedList* ll, void* data, int n){
  Node* nn = new_node(data);
  if (!nn) return;
  int i=0;
  Node* p = ll->head;
  if (!p) ll->head = nn;
  else {
    while (p->next && i < n){
      p = p->next; i++;
    }
    nn->next = p->next;
    p->next = nn;
  }
}
```

Linked Lists: Operations - insert

```
void insert_in_order(LinkedList* ll, void* data, int(*comp)(void*,void*)){
  Node* n = new_node(data);
  if (!n) return;
  Node* p = ll->head;
  if (!p || comp(data, p->data)<0){
    n->next = p;
    ll->head = n;
  }
  else {
    while (p->next && comp(data, p->next->data)>0) p=p->next;
    n->next = p->next;
    p->next = n;
  }
}
```

Linked Lists: Operations - remove

```
void* remove_from_front(LinkedList*ll){
  void* result;
  Node* p = ll->head;
  if (!p) return NULL;
  result = p->data;
  ll->head = p->next;
  free(p);
  return result;
}
```

```
void* remove_from_back(LinkedList*ll){
  void* result;
  Node* p = ll->head;
  if (!p) return NULL;
  if (!(p->next)){
    result = p->data;
    ll->head = NULL;
    free(p);
  }
  else {
    while (p->next->next) p=p->next;
    result = p->next->data;
    free(p->next);
    p->next = NULL;
  }
  return result;
}
```

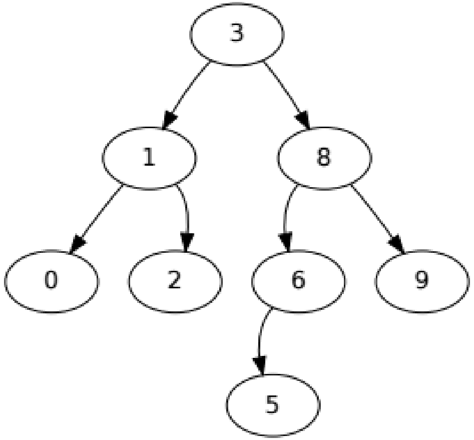
Linked List vs Arrays - operations

- Time complexity:
Operation	**Linked List**	**Array**
Indexing	O(n)	O(1)
Insert at front	O(1)	O(n)
Insert at back	O(n)	O(1)
Remove from front	O(1)	O(n)
Remove from back	O(n)	O(1)
- Other aspects:
Aspect	**Linked List**	**Array**
Extensibility	dynamic size	fixed size: expansion is costly
Shifting	not required	some operations (discuss)
Random access	inefficient	efficient
Sequential access	slow	fast (discuss)
Memory use	efficient	inefficient for large arrays and few data

Binary Trees

- Binary Tree: dynamic data structure where <u>each node</u> has <u>at most</u> two children
- A binary search tree is a binary tree with ordering among its children
  - all elements in the left subtree are assumed to be ”less” than the root element
  - and all elements in the right subtree are assumed to be "greater" than the root element
- Example:

```
struct tnode{
  void* data; /* payload */
  struct tnode* left;
  struct tnode* right;
};
struct tree{
  struct tnode root;
}
```



Binary Trees

- The operation on trees can be framed as recursive operations.
    - Traversal (printing, searching):
        - pre-order: root, left subtree, right subtree
        - inorder: left subtree, root, right subtree
        - post-order: right subtree, right subtree, root

- Add node:

```
struct tnode* addnode(struct tnode* root, int data){
  if (root==NULL){ /* termination condition */
    /* allocate node and return new root */
  }
  else if (data < root->data) /* recursive call */
         return addnode(root->left, data);
       else
         return addnode(root->right, data);
}
```

Stack

- A structure that stores data with restricted insertion and removal:
  - insertion occurs from the top exclusively: push
  - removal occurs from the top exclusively: pop
- ```
  typedef struct Stack Stack;
  Stack* new_stack(int size);
  void* pop(Stack* q);
  void push(Stack* q, void* data);
  ```
- may provide `void* top(void);` to read last (top) element without removing it
  ```

Stack as an Array

stack.h stack_ar.c test1.c

```
struct Stack{
  int capacity;
  void** buffer;
  int top;
};
```

- Stores in an array buffer (static or dynamic allocation)
- insert and remove done at end of array; need to track end
- ```
  Stack* new_stack(int size){
    Stack* result = (Stack*)calloc(1,sizeof(Stack));
    result->capacity = size;
    result->buffer = (void**)calloc(size, sizeof(void*));
    return result;
  }
  ```

```
void push(Stack* s, void* data){
  if (s->top < s->capacity)
    s->buffer[s->top++] = data;
}
```

```
void* pop(Stack* s){
  if (s->top > 0)
    return s->buffer[--(s->top)];
  else return NULL;
}
```
```

Stack as a Linked List

ll.h ll.c stack.h stack_ll.c

```
struct Stack{
  LinkedList* buffer;
};
```

- Stores in a linked list (dynamic allocation)
- “Top” is now at front of linked list (no need to track)
- ```
  Stack* new_stack(int size){ /* size is not needed */
    Stack* result = (Stack*)calloc(1,sizeof(Stack));
    result->buffer = new_linked_list();
    return result;
  }
  ```

```
void push(Stack* s, void* data){
  insert_at_front(s->buffer, data);
}
```

```
void* pop(Stack* s){
  return remove_from_front(s->buffer);
}
```
```

Queue

- Opposite of stack:
  - first in: enqueue
  - first out: dequeue
  - Read and write from opposite ends of list
- Important for:
  - UIs (event/message queues)
  - networking (Tx, Rx packet queues)
  - :
- Imposes an ordering on elements
-
```
typedef struct Queue Queue;
Queue* new_queue(int size);
void* dequeue(Queue* q);
void enqueue(Queue* q, void* data);
```

Queue as an Array

queue.h   queue_ar.c   test1.c

- Stores in an array buffer (static or dynamic allocation);
- Elements added to rear, removed from front
  - need to keep track of front and rear: `int front=0, rear=0;`
  - or, track the front and number of elements: `int front=0, count=0;`

```
struct Queue{
   int capacity;
   void** buffer;
   int front;
   int count;
};
```

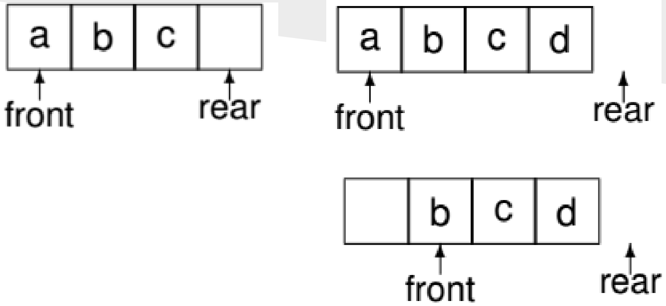
```
Queue* new_queue(int size){
 Queue* result = (Queue*)calloc(1,sizeof(Queue));
 result->capacity = size;
 result->buffer = (void**)calloc(size,sizeof(void*));
 return result;
}
```

```
void enqueue(Queue* q, void* data){
 if (q->count < q->capacity){
  q->buffer[q->front+q->count] = data;
  q->count++;
 }
}
```

```
void* dequeue(Queue* q){
   if (q->count > 0) {
     q->count--;
     return q->buffer[q->front++];
   }
   else return NULL;
}
```

Queue as an Array

queue.h queue_arr.c test1.c



- Let us try a queue of capacity 4:
  - enqueue a, enqueue b, enqueue c, enqueue d
  - queue is now full.
  - dequeue , enqueue e: where should it go?
- Solution: use a circular (or ring) buffer
  - 'e' would go in the beginning of the array
- Need to modify enqueue and dequeue:

```
void enqueue(Queue* q, void* data){
 if (q->count < q->capacity){
  q->buffer[q->front+q->count %
                q->capacity] = data;
  q->count++;
 }
}
```

```
void* dequeue(Queue* q){
  void* result = NULL;
  if (q->count > 0) {
    q->count--;
    result=q->buffer[q->front++];
    if (q->front == q->capacity)
      q->front = 0;
  }
  return result;
}
```

Queue as a Linked List [ll.h] [ll.c] [queue.h] [queue_ll.c]

```
struct Queue{
  LinkedList* buffer;
};
```

- Stores in a linked list (dynamic allocation)
- ```
  Queue* new_queue(int size){
    /* size is not needed*/
    Queue* result = (Queue*)calloc(1,sizeof(Queue));
    result->buffer = new_linked_list();
    return result;
  }
  ```

```
void enqueue(Queue* q, void* data){
  insert_at_back(q->buffer, data);
}
```

```
void* dequeue(Queue* q){
  return remove_from_front(q->buffer);
}
```
```

Example: Postfix Evaluator

- Stacks and queues allow us to design a simple expression evaluator
- Prefix, infix, postfix notation:
  - operator before, between, and after operands, respectively
  - Infix
    - A + B
    - A * B - C
    - (A + B) * (C - D)
  - Prefix
    - + A B
    - - * A B C
    - * + A B - C D
  - Postfix
    - A B +
    - A B * C
    - A B + C D - *
  - Infix more natural to write, postfix easier to evaluate

Example: Postfix Evaluator

```
float pf_eval(char* exp){
  Stack* S = new_stack(0);
  while (*exp){
    if (isdigit(*exp) || *exp=='.'){
      float* num; num = (float*)malloc(sizeof(float));
      sscanf(exp, "%f", num);
      push(S, num);
      while(isdigit(*exp) || *exp=='.') exp++; exp--;
    }
    else if (*exp!= ' ') {
      float num1 = *(float*)pop(S), num2 = *(float*)pop(S);
      float* num; num = (float*)malloc(sizeof(float));
      switch (*exp){
        case '+': *num = num1+num2; break;
        case '-': *num = num1-num2; break;
        case '*': *num = num1*num2; break;
        case '/': *num = num1/num2; break;
      }
      push(S, num);
    }
    exp++;
  }
  return *(float*)pop(S);
}
```

Hash Table

- Hash tables (hashmaps) an efficient data structure for storing dynamic data.
- commonly implemented as an array of linked lists (hash tables with chaining).
- Each data item is associated with a key that determines its location.
  - Hash functions are used to generate an evenly distributed hash value.
  - A hash collision is said to occur when two items have the same hash value.
  - Items with the same hash keys are chained
  - Retrieving an item is O(1) operation.
- Hash function: map its input into a finite range: hash value, hash code.
  - The hash value should ideally have uniform distribution. why?
  - Other uses of hash functions: cryptography, caches (computers/internet), bloom filters etc.
  - Hash function types:
    - Division type
    - Multiplication type
    - Other ways to avoid collision: linear probing, double hashing.

Hash Table

- ```
  struct Pair{
    char* key;
    void* data;
  };
  ```
- ```
  struct HashTable{
    LinkedList* buckets;
    int capacity;
  };
  ```
- ```
  unsigned long int hash(char* key);
  HashTable* new_hashtable(int size);
  int is_empty_ht(HashTable* ht);
  int length(HashTable* ht);
  void insert(HashTable* ht, Pair* p);
  void* remove(char* key);
  void* retrieve(char* key);
  ```
  ```

Hash Table

```
HashTable* new_hashtable(int size){
  HashTable* result = (HashTable*)calloc(1, sizeof(HashTable));
  result->size = size;
  result->buckets = (LinkedList*)calloc(size, sizeof(LinkedList));
  return result;
}

int is_empty_ht(HashTable* ht){
  int i;
  for (i=0; i < ht->size; i++)
    if (!is_empty(&(ht->buckets[i])))
      return 0;
  return 1;
}

int length(HashTable* ht){
  int i, result=0;
  for (i=0; i < ht->size; i++)
    result += size(&(ht->buckets[i]));
  return result;
}
```

Hash Table

```
unsigned long int hash(char* key){
  /* any good hashing algorithm */
  const int MULTIPLIER = 31;
  unsigned long int hashval = 0;
  while (*key)
    hashval = hashval * multiplier + *key++;
  return hashval;
}

void insert(HashTable* ht, Pair* p){
  if (retrieve(ht, p->key)) return NULL;
  int index = hash(p->key) % ht->capacity;
  insert_at_front(&(ht->buckets[index]));
}

void* remove(char* key){
  /* since we do not have a ready supporting ll function we'll go low level */
}

void* retrieve(char* key){
  /* since we do not have a ready supporting ll function we'll go low level */
}
```