Lec_23.md

Lecture 23 - Linked Lists

Nov. 03/2020

Announcements:

Final Summative Assessment

- **2.5 hour** final exam
- **Monday, December 14, 2020** starting at **9:30 am EST**
- **Synchronous Exam**
  - Everyone writes the exam at the same time
- Covers all the lecture material from start of term until end of term
  - Covers all lab assignments
- **Open textbook** and **open notes**, but no **calculators, compilers, IDEs**
- Will use **Quercus Quizzes**
- Past finals posted on Quercus
  - Good practice, *coverage varies*

Quiz 4

- **45 minute** quiz (longer than usual)
- **Friday, November 6, 2020**
  - Take it any time in the day
- If you leave the quiz you can't go back to it
- Cover all the **lecture material** from start of semester until **end of this lecture**
- Covers Lab 1 to Lab 3
- **Open textbook** and **open notes**

Linked Lists

```
class listNode{
public:
  int key;
  listNode* next;
};

listNode node;



class linkedList{
public:
  listNode* head;
};

linkedList myList;
```

Inserting a value at Tail

Given a new node, want to insert the node so it is the *last* node in the list

- Must start traversing **linked list** at head
  - Use `tptr` to traverse the **linked list**
    - Stopping when `tptr->next` is `nullptr`

```
void linkedList::insertAtTail(listNode* nptr) {
  listNode* tptr = head;
  while (tptr->next != NULL)
    tptr = tptr->next;
  tptr->next = nptr;
  nptr->next = NULL;
}
```

Notes:

1. `while (tptr->next != NULL)` and `tptr= tptr->next;`

- Traverse through the list using `tptr`
- Stop when `tptr->next` is `NULL`
  - This means `tptr` points to the last node in the **linked list** or the **tail** node

2. `tptr->next = nptr;`

- Set new **tail** node to `nptr`

3. `nptr->next = NULL;`

- Set the `next` field for the **tail** node to null (new **tail** node)

4. What if the **linked list** is empty?

- e.g. the **head** node is `null` ?
  - Then you get a segmentation fault when running `tptr->next` as `tptr` currently points to `head` , and `head` has no value
- So we need to protect against this case

```
void linkedList::insertAtTail(listNode* nptr) {
  if (head == NULL) {
    head = nptr;
    nptr->next = NULL:
  }
  else {
    listNode* tptr = head;
    while (tptr->next != NULL)
      tptr = tptr->next;
    tptr->next = nptr;
    nptr->next = NULL;
  }
}
```

Notes:

1. `if(head == NULL){ ... }`

- Check if **linkedlist** is empty
- Prevents the seg fault from earlier

Inserting a value in the Middle

Given a **key** k, located a node with this key and then insert the new node pointed to by `nptr` after the located node

- Assuming k exists
  - You can define behaviour if k DNE
- Need to *break* the **linked list** chain
  - Make `nptr->next` point to the element after insertion point
  - Make `tptr->next` point to the inserting node, `nptr`

```
void linkedList::insertInMiddle(int k, listNode* nptr) {
  listNode* tptr = head;
  while (tptr != NULL) {
    if (tptr->key == k) break;
      tptr = tptr->next;
  }
  nptr->next = tptr->next;
  tptr->next = nptr;
}
```

Notes:

1. `while (tptr != NULL){ ... }`

- Traverse list until you find node with `key` k

2. `nptr->next = tptr->next;`

- Assign the `nptr->next` to point to `tptr->next;`

3. `tptr->next=nptr;`

- Assign the `tptr->next` to point to `nptr`

4. 2 and 3 *must* be executed in this order

- If 3 is executed first, then the value of `tptr->next` will be lost

Deleting a Node

Given a **key** k, locate a node with this key and delete it

- Idea: Use *two* traversing pointers, `tptr` and `pptr`
  - `pptr` lags `tptr` by 1 node

```
void linkedList::deleteNode(int k) {
  listNode* tptr = head;
  listNode* pptr = NULL;
  while (tptr != NULL) {
    if (tptr->key == k) break;
    pptr = tptr;
    tptr= tptr->next;
  }
  pptr->next = tptr->next;
  delete tptr;
}
```

Notes:

1. `ptpr->next = tptr->next;`

- Since `ptpr` lags `tptr` by 1 node, this detaches/removes one node from the list

2. Tons of edge cases for this lmao

- `head` is null (list empty)
- `key` not found
- trying to delete `head` node

Addressing `key` not found:

```
void linkedList::deleteNode(int k) {
  listNode* tptr = head;
  listNode* pptr = NULL;
  while (tptr != NULL) {
    if (tptr->key == k) break;
    pptr = tptr;
    tptr= tptr->next;
  }
  if(tptr == NULL) return; //fix
  pptr->next = tptr->next;
  delete tptr;
}
```

Addressing `head` is null, or the list is empty:

```
void linkedList::deleteNode(int k) {
  if(tptr == NULL) return; //fix
  listNode* tptr = head;
  listNode* pptr = NULL;
  while (tptr != NULL) {
    if (tptr->key == k) break;
    pptr = tptr;
    tptr= tptr->next;
  }
  if(tptr == NULL) return;
  pptr->next = tptr->next;
  delete tptr;
}
```

Notice that this covers the `key` *not found* case above, so we can replace this implementation with one that covers both cases, in fewer lines of code

```
void linkedList::deleteNode(int k) {
  if(tptr == NULL) return; //kept fix here, this covers fix below
  listNode* tptr = head;
  listNode* pptr = NULL;
  while (tptr != NULL) {
    if (tptr->key == k) break;
    pptr = tptr;
    tptr= tptr->next;
  } //deleted fix below
  pptr->next = tptr->next;
  delete tptr;
}
```

Addressing deleting the `head` node

```
void linkedList::deleteNode(int k) {
  if(tptr == NULL) return;
  listNode* tptr = head;
  listNode* pptr = NULL;
  while (tptr != NULL) {
    if (tptr->key == k) break;
    pptr = tptr;
    tptr= tptr->next;
```

```
  }
  if (tptr == head) {
    head = head->next;
    delete tptr;
    return;
  }
  pptr->next = tptr->next;
  delete tptr;
}
```

Notes:

1. `if (tptr == head) { ... }`

- Adds a case for if the node we are deleting is the head node

listNode Class Definition

```
class listNode {
  private:
    int key;
    listNode* next;
  public:
    listNode();
    listNode(int k);
    listNode(int k, listNode* n);
    listNode(const listNode& other);
    ~listNode();
    int getKey() const;
    listNode* getNext() const;
    void setKey(int k);
    void setNext(listNode* n);
    void print() const;
};
```

listNode Accessors/Mutators/Print

```
int listNode::getKey() const {
  return (key);
}
listNode* listNode::getNext() const {
  return (next);
}
void listNode::setKey(int k) {
  key = k;
}
void listNode::setNext(listNode* n) {
  next = n;
}
void listNode::print() const {
  cout << "(" << key << "," << next << ") ";
}
```

listNode Constructors

```
listNode::listNode() {
  key = 0;
  next = NULL;
}
listNode::listNode(int k) {
  key = k;
```

```
  next = NULL;
}
listNode::listNode(int k, listNode* n) {
  key = k;
  next = n;
}
listNode::listNode(const listNode& other) {
  key = other.key;
  next = other.next;
}
listNode::~listNode() {
  //Nothing to do
}
```

Notes:

1. `listNode::listNode(const listNode& other)`

- Method shallow on purpose
  - Design decision

2. `listNode::~listNode()`

- Method shallow on purpose
  - When one node is deleted
    - Should all the *next* nodes also be deleted?
      - Would be a **deep** implementation
    - Or rather keep *next* nodes
      - **Shallow** implementation
  - You have control: you can define if you want **shallow** or **deep** behaviour

linkedList Class Definition

Moving on to the **linked list** class definition

```
class linkedList {
  private:
    listNode* head;
  public:
    linkedList();
    linkedList(const linkedList& other);
    ~linkedList();
    linkedList & operator=(const linkedList& rhs);
    bool insertKey(int k);
    bool deleteKey(int k);
    bool keyExists(int k);
    void print() const;
};
```

linkedList Constructors

```
linkedList::linkedList(){
  head = NULL;
}

linkedList(const linkedList& other){
  listNode* ptr = other.head;
  listNode* nptr = NULL;
  head = NULL;
  while(ptr != NULL) {
    nptr = new listNode(ptr->getKey());
```

```
    // insert *nptr at end of list
    ptr = ptr->getNext();
  }
}
```

Notes:

1. `linkedList::linkedList()`

- Create a new **linked list**, with head `NULL`

2. We want the **copy constructor** to have a **deep** implementation

- If we copy a list, we want to keep all the *next* nodes

3. `linkedList(const linkedList& other)`

-