Data Structures and Algorithms

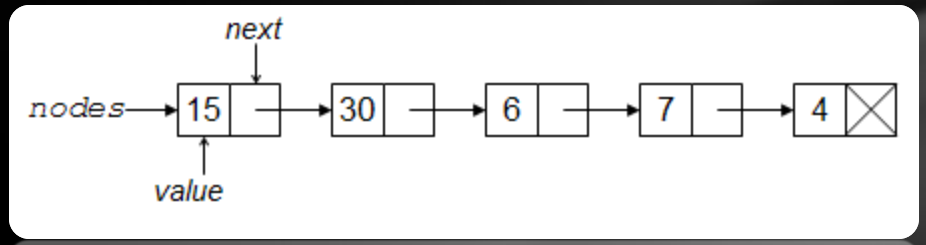
Lecture 6

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

-Bill Gates

Lists

- Lists are used to store data in fragmented memory areas
- They are made up of nodes that hold data and references to other nodes
- They have advantages and disadvantages over arrays and they are very useful in certain application domains



Example of a singly linked list with integer values

Lists (cont'd)

Pros

- Incremental memory allocation – allocate nodes as you need them, delete the nodes as you're done with them
- Continuous memory area is not required for large data since the nodes can be allocated anywhere in the heap

Cons

- No random access! In order to get to a node, you need to iterate through all the nodes in the list before the target node
- Memory overhead for references to nodes (each node holds a reference to the next one) which is even larger for doubly linked lists

Lists (cont'd)

Arrays

- Continuous memory area
- Fixed allocated size at creation
- Random access inside the array
- Difficult to insert & delete elements
- All allocated memory is used to hold data

Lists

- Fragmented memory area
- Memory allocation is done incrementally
- Only sequential access through the list
- Easy to insert & delete elements
- Memory overhead used to store references to the next node

Lists (cont'd)

- There are two types of lists:
  - Singly linked lists – where a node only has a reference to the next node (there is no way to return to the previous node)
  - Doubly linked lists – where a node holds a reference to both the next and the previous node

```
struct node{
    int data;
    struct node * next;
};
```

```
struct node{
    int data;
    struct node * previous;
    struct node * next;
};
```

Review: what is the size of these two structures?

Lists - iterating

- For a list, we usually hold a pointer to the first and maybe the last node in that list
- Iterating implies moving through the list from one node to the next, processing it in some way, until we reach the last node, or a NULL next pointer
- Iteration is obviously done with a loop construct.

Example:

```
struct node *first = …;
struct node *current_node = first;
while(current_node != NULL){
    process_node(current_node);
    current_node = current_node->next;
}
```

Exercise: rewrite the above example using a for loop.

Lists - inserting

Inserting an element in a list is done by allocating memory for a new node and then simply re-doing the connections between nodes;

Example:

```
void insert_value_after_node(int value, struct node* current_node){
    struct node * new_node = (struct node*)malloc(sizeof(struct node));
    new_node->value = value;
    new_node->next = current_node->next;
    current_node->next = new_node;
}
```

Exercise: do the same thing for a doubly linked list.

Lists - deleting

Deleting an element from a list is done by re-doing the connections between nodes and then freeing the unused memory.

Example:

```
void remove_next_node(struct node* current_node){
    struct node * old_node = current_node->next;
    current_node->next = old_node->next;
    free(old_node);
}
```

Exercise: do the same thing for a doubly linked list.

Lists - practice

- Search and display the indexes of all elements in a list that are equal to a specified value.

New tool: valgrind – used to analyze if your program frees all allocated memory and if you are trying to read or write outside allocated memory.

- Sorting a list: which is the best algorithm for sorting a singly linked list? Which is the easiest to implement? How about a doubly linked list?

Thank you!