Lecture 14: Implementing linked data structures
Linked data structures

• Linked data structures – “self-referential” types

struct listnode {
    DataType data;
    /* DataType is defined elsewhere */
    struct listnode *next;
    /* a pointer to next node */
};
Question: what is a linked list?

• Answer: a sequence of zero or more nodes, with each node pointing to the next one
• Need: a pointer to the first node – head
  – Often this pointer (i.e., head) is considered “the list”
  – head might be NULL – just indicates an empty list
  – We use NULL to denote that a pointer is not pointing anywhere
Linked data structures

• Made up of nodes, and links between nodes

• As purpose is data storage/retrieval, also contains information field(s) inside nodes

• Simplest is a linear linked list with single links
  – Key is to define a node type to hold info and a link:

```c
struct ListNode {
    DataType data; /* e.g., int, char*, ... */
    ListNode *next;
};
```

  – `next` is for pointing at the next node in the list

  – If `next == NULL` then it is the last node in the list
How to search a list

• Idea: *return pointer to node that contains equal data, or return NULL if list has no equal data*

• Basic algorithm:
  
  *declare local node pointer - call it current*
  
  *point current at first node in list (i.e., head)*
  
  *while (current points to non-null node)*

    *if (current is pointing to a node with the searched data):* 
      
      *return current*

    *else:* 

      *advance current to current.next*

  
  *return NULL if get this far*
Traversing a list in general

• Search strategy typifies link-hopping activity
  start by pointing at first node
  while not pointing at NULL:
    process the node
    reset pointer to the node’s next field

  – Same idea to print list, or anything that requires traversing the list

• Always consider special cases: first node, last node, empty list, just one node in the list, …
  – Depends on the type of operation
Inserting to a linked list

• Assume that we want to keep the list sorted,
  – so we need to insert the new node to the correct place

• In some cases that may mean inserting the new node before the node at the head of the list
  – Which means that we will need to update the value of the head pointer

• In order to update the head pointer, we need to pass pointer to the head pointer
  – So we will pass a pointer to a pointer to the insert function!
Inserting to a sorted list

create new node
   aim a pointer named new at it
   set its data field to new data
   set its next field to NULL

if the head is NULL or new data is smaller than head’s data:
   set next field of new to head
   point head at new
else
   traverse the list to find the place to insert
/* traversing the list to find the place to insert: */
use two pointers: current and previous, initialize to head
while current->data < new data and current->next != NULL
    previous = current;
    current = current->next;
if (current->data == data)
    Print: the value is already in the list
else if (current->data < data)
    current->next = new; /* insert to the end of the list */
else if (current->data > data)
    new->next = current; /* insert before the current */
previous->next = new;
Deleting an item from a linked list

• Deleting an item from the list may require us to modify the head pointer
  – The deleted item could be the one pointed by the head pointer

• In order to update the head pointer, we need to pass pointer to the head pointer
  – So we will pass a pointer to a pointer to the delete function!
Algorithm for deleting a data item from a list

check if the data item is in the list first
  if not return
check if the data item is the one pointed by the head pointer
  update the head pointer
  and free the node
else /* traverse the list to find the item */
use two pointers: current and previous, initialize to head
while (current->data != data)
  previous = current;
  current = current->next;
/* remove the current node */
previous->next = current->next;
free(current);
Other linked structures

There are more elaborate linked structures:

• Doubly-linked lists: Nodes with 2 links: previous and next
  • Easy *reverse* traversal, insertion *before* a node, …
  • But 2 links to worry about for insert, remove, …

• Circular lists – last points to first (and first points to last for 2-way circular list)

• Choice depends on problem and efficiency (more to come about that too)
Implementing data structures

Using the C structures and pointers we can implement many data structures

• Queues: A data structure where we always insert to the head of the list and always remove from the tail of the list
  – First In First Out: FIFO storage

• Stacks: A data structure where we always insert and remove from the head of the list
  – Last In First Out: LIFO storage

• Trees: Where each node can have arbitrary children but each node has one parent

• Graphs: Where each node can arbitrary number of adjacent nodes
Miscellaneous C features related to data storage

• Unions
  – Sharing the same memory for storing data of different types (and sizes)

• Bit-level operations
  – Accessing and manipulating bits for more efficient use of memory
Unions

- Hold different data types/sizes (at different times)
- e.g., define union to hold an int or a double:

```c
union aValue{
    int x;
    double d;
} u, *up;
/* u is a union, up can point to one */
```

- Access x or d by u. or up-> just like structures

- sizeof u is size of largest field in union
  - Equals sizeof(double) in this case

- Often wrap in a structure, with key to identify type
  ```c
  struct{union aValue v; char key;} wrapper;
  ```
Using bits (when space is precious)

• Consider: 4-byte int means 32 bits of information!
  – Could treat as 32 meaningful values
    • each bit is either 0 or 1

• Use bit operators & and | to set or test
  - a & b \rightarrow \text{bits on where both a bits and b bits are on}
  - a | b \rightarrow \text{bits on where either a bits or b bits are on}

• e.g., i = 3 \rightarrow \text{first 30 bits off, last 2 bits on}
  - Set flags: \text{hot}=4, \text{awake}=2 \text{ and } \text{alive}=1, \text{then}
    - i\&awake \text{ and } i\&alive \text{ both true, but } i\&hot \text{ is false}

• Also: can shift bits left \ll or right \gg
Bit-fields – using bits more clearly

- Use a special syntax to define a structure
  - Specify width (in bits) of each field with \( w \)
    ```
    struct flags {
      unsigned int isAlive : 1;
      unsigned int isAwake : 1;
      unsigned int isHot : 1;
    } i = {1,1,0};
    ```

- Now use field names like any other structure
  - `i.isAlive` and `i.isAwake` both true, `i.isHot` is false
  - `i.isAwake = 0; /* i just fell asleep */`
  - cannot do `&i.isAlive` (\&i still okay)

- See `.../demos/getFlags.chf` (uses flags.h)