ADTs and vectors, towards linked lists

- **tvectort** is a class-based implementation of a lower-level data type called an array
  - **tvectort** grows dynamically (doubles in size as needed) when elements inserted with `push_back`
  - **tvectort** protects against bad indexing, arrays don’t
  - **tvectort** supports assignment: `a = b`, arrays don’t
- As an ADT (abstract data type) vectors support
  - Constant-time or O(1) access to the k-th element
  - Amortized O(n) storage/time with `push_back`
    - Total storage used in n-element vector is approx. 2n, spread over all accesses/additions
  - Adding a new value in the middle of a vector is expensive, O(n) because shifting required

Linked lists

- Low-level (concrete) data structure, used to implement higher-level structures
  - Used to implement sequences/lists (see `CList` in Tapestry)
  - Basis of common hash-table implementations (later)
  - Similar to how trees are implemented, but simpler
- Linked lists as ADT
  - Constant-time or O(1) insertion/deletion from anywhere in list, but first must get to the list location
  - Linear or O(n) time to find an element, sequential search
- Good for sparse structures: when space is scarce, allocate exactly as many list elements as needed

Linked lists and lists, CDT and ADT

- As an ADT what is a list and what operations are supported?
  - ( ) or (x, ( ) ),
  - A list is empty, or contains an element and a list
  - Access head/first and tail/rest of list, see `CList` for details
- As CDT (concrete data type)

```c
struct Node
{
    string info;
    Node * next;
    Node(const string &s, Node * link)
    { info(s), next(link) }
};
```

- How can we add a new Node? How would a constructor help?

Processing linked lists

- Add words to the front of a list (draw a picture)
  - What about adding to the end of the list?

```c
struct Node
{
    string info;
    Node * next;
    Node(const string s, Node * link)
    { info(s), next(link) }
};
// ... declarations here
Node * list = NULL;
while (input >> word)
{    list = new Node(word, list);
}
```
**Header (aka dummy) nodes**

- Special cases in code lead to problems
  - The special cases permeate the code, hard to reason about correctness,
  - Avoid special cases when trade-offs permit
    - Space, time trade-offs
- In linked lists it’s useful to have a header node, the empty list is not NULL/0, but a single “blank” node
  - Every node has a node before it, avoid special code for empty lists
  - Header node is skipped by some functions, e.g., count the values in a list
  - What about a special “trailing” node?
  - What value is stored in the header node?

**Variations: doubly and circularly linked**

- In singly-linked lists, need pointer-to-node before to remove a node from a list, why?
  - How do header nodes help? (See `linkcount.cpp`)
- Move forward/backwards in a doubly linked list, what needs to be added to `Node` declaration?
  - Downside?

**Circularly linked list**

- If the last node points to NULL/0, the pointer is “wasted”
  - Keep pointer to last node, but:
    - How is first node accessed?
    - How is last node accessed?
    - What does a one node list look like?
  
  ```
  // standard linked list
  while (list != NULL) {
    Process(list);
    list = list->next;
  }
  // circularly linked
  Node * first = list->next;
  while (list != NULL) {
    Process(list);
    list = list->next;
  }
  while (first != list);
  ```

- Potential problems? Failures?

**Idiomatic linked list functions**

- Pass in a list, return *altered* list
  - Needed when no header node used, can use header node or pass list by reference
  ```
  list = Change(list, /* other params */);
  Node * Change(Node * list, const string& key) {
    if (list != 0) {
      list->next = Change(list->next, key);
      if (list->info == key) return list->next;
      else return list;
    }
  }
  ```

- How can we reason about this code?
  - Empty list, one-node list, two-node list, n-node list
  - Similar to proof by induction
Idiomatic list/loop processing

- Visit all nodes once, e.g., count them

```c
int Size(Node * list) {
    count = 0;
    while (list != 0) {
        count++;
        list = list->next;
    }
    return count;
}
```

- Print nodes, changes? Append “s” to all strings in list, changes?

Idiomatic list/recursive processing

- Visit all nodes once, e.g., count them

```c
int Size(Node * list) {
    if (list == 0) return 0;
    return 1 + Size(list->next);
}
```

- Base case is almost always empty list – NULL/0 node
  - Must return correct value, perform correct action
  - Recursive calls use this value/state to anchor recursion
  - Sometimes one node list also used, two “base” cases

- Recursive calls make progress towards base case
  - Almost always using list->next as argument

Recursion and linked lists

- Print nodes in reverse order
  - Print all but first node and...
    - Print first node before or after other printing?

```c
void Print(Node * list) {
    if (list != 0) {
        // Print first node before or after other printing?
    }
}
```

Reverse list: (a, b, c, d) to (d, c, b, a)

```c
Node * Reverse(Node * list) {
    // post: return list reversed - list changed,
    //       new nodes NOT created
    if (list != 0) {
        Node * rest = Reverse(list->next);
        return ;
    }
    return ;
}
```
Efficient reverse, generalize?

```c
Node * ReverseAux(Node * list, Node * rev)
    // pre: rev is what's been reversed so far
    // post: return list reversed
    {
        if (list != 0)
        {
            Node * temp = list->next;
            list->next = rev;
            return ReverseAux(temp, list);
        }
        return rev;
    }

Node * Reverse(Node * list)
    {
        return ReverseAux(list, 0);
    }
```

From recursion to iteration (and back?)

- In `ReverseAux`, two important concepts
  - What is invariant property of `rev`? (see precondition)
  - Recursive call is last call of function, tail recursion
    - Tail recursion is easy to “undo” into iteration (why bother?)

- Rewrite iteratively
```c
Node * Reverse(Node * list)
    // post: return list reversed
    {
        // invariant: rev is reverse of list so far
        Node * rev = 0;
        while (list != 0)
        {
            // what are cases for a and b?
            Node * temp = list->next;
            list->next = rev;
            rev = list;
            list = temp;
        }
        return rev;
    }
```

Merge two sorted lists

```c
Node * Merge(Node* a, Node* b)
    // pre: a and b are sorted
    // post: return new sorted list, containing all values
    // in a and b
    {
        // what are cases for a and b?
        Node * temp = list->next;
        list->next = rev;
        return ReverseAux(temp, list);
    }
```

- Why bother merging two sorted lists?

Hybrid structures: vectors and lists

- We can store word/counts in a vector, see `wordcount.cpp`
- We can store word/counts in linked list, see `linkcount.cpp`
  - Advantages of approaches?
  - Alternatives within an approach? Between?

- What about a vector of linked lists?
  - One linked list per letter of the alphabet: ‘a’ – ‘z’
  - Why use vector of linked lists rather than linked list of
    linked lists?
  - What about a vector of vectors? Possible? Drawbacks?
  - What about more than 26 linked lists, 52? Ten-thousand?