ADTs and vectors, towards linked lists

- `tvector` is a class-based implementation of a lower-level data type called an array
  - `tvector` grows dynamically (doubles in size as needed) when elements inserted with `push_back`
  - `tvector` protects against bad indexing, arrays don’t
  - `tvector` 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 linear or O(n) storage/time with `push_back`
    - Total storage used in n-element vector is approx. 2n, spread over all accesses/additions (why?)
  - Adding a new value in the middle of a vector is expensive, linear or O(n) because shifting required

Pointer basics

- Memory is allocated dynamically at runtime from the heap
  - Contrast to statically allocated at compile time
    - Static variables take up space on the runtime stack, program executable may be large as a result
    - ```
      void foo(const Date& d)
      {
        int y;
        tvector<int> scores(20);
      }
    ```
    - Scores isn’t 20x bigger than y, why?
- Pointers reference memory, a pointer is different from the object it points to. There is a pointer and a pointee.

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
  - Like a film or video tape: splicing possible, access slow
- Good for sparse structures: when space is scarce, allocate exactly as many list elements as needed

Pointer/Pointee confusion?

- Pass-by-value, can we change the parameter?
  - ```
    void doStuff(Date * d)
    {
      d = new Date();
    }
    ```
  - ```
    void doStuff2(Date * d)
    {
      *d += 1;
    }
    ```
  - Date * flagDay = new Date(6,14,2001);
  - doStuff(d);
  - cout << *d << endl;
- In case things aren’t confusing enough
  - `const Date * d;` // pointee is constant
  - `Date * const d;` // pointer is constant
Linked list applications

- Remove element from middle of a collection, maintain order, no shifting. Add an element in the middle, no shifting
  - What’s the problem with a vector (array)?
  - Emacs visits several files, internally keeps a linked-list of buffers
  - Naively keep characters in a linked list, but in practice too much storage, need more esoteric data structures

- What’s \( (3x^6 + 2x^4 + x + 5) + (2x^4 + 5x^3 + x^2 + 4x) \)?
  - Store as \((3, 0, 2, 0, 1, 5)\) and \((0, 2, 5, 1, 4, 0)\)
  - Store as \((3, 5), (2, 3), (1, 1), (5, 0)\) and _________?
  - Most polynomial operations sequentially visit terms, don’t need random access, do need “splicing”

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)

  ```
  struct Node               Node * p = new Node();
  {                           p->info = "hello";
      string info;            p->next = NULL;  // 0
      Node * next;
  }
  ```

Processing linked lists

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

  ```
  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
Node * first = list->next;
while (list != NULL) {
  Process(list);
  list = list->next;
}
```

```
// circularly linked
Node * first = list->next;
while (list != NULL) {
  Process(list);
  list = list->next;
}
```

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

```
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;
  }
}
```

Idiomatic list/loop processing

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

```
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(list->next);
      printf("%d\n", list->data);
    }
  }
  ```

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 new Node(list->data, rest);
  }
  return NULL;
}
```  

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?