Pointer-Based Data Structures

- **Application-oriented datatypes**: We have seen use of structures for points, lines, times, days…

- **Goal**: Variables for sequences, sets, dictionaries, graphs, …

- **Problem**: Number of elements not known in advance, and changing during run-time.
  - Fixed-size arrays: either too limiting, or wasting memory
  - Variable-size arrays: slow operations

- **Solution Idea**:
  - elements are kept in a “wrapper” structures on the heap
  - wrappers contain pointers linking all elements together

- **Simplest Instance**: singly-linked lists

  *Use*: sequences, stacks; sets, dictionaries

---

Linked Lists — Creation

typedef struct CharListNodeStruct {
  char data;
  struct CharListNodeStruct * nextPtr; /* struct label necessary */
} CharListNode;

typedef CharListNode * CharList;

CharList newCharListNode(char c, CharList rest) {
  CharList r = malloc(sizeof(CharListNode));
  if (r == NULL) fprintf(stderr,"newCharListNode: out of memory\n");
  else { r->data = c; r->nextPtr = rest; } return r;
}

- The contents of a variable of type CharList is a list
- The empty list is the value NULL
- A non-empty list is a pointer to a malloced list node in the heap

---

Linked Lists — Insertion

typedef struct CharListNodeStruct {
  char data;
  struct CharListNodeStruct * nextPtr; /* struct label necessary */
} CharListNode;

typedef CharListNode * CharList;

/* Insertion changes the argument list — pass by reference necessary */
void insert(CharList * p, char val) {
  if (p == NULL || val <= (*p)->data) { *p = newCharListNode(val, *p); } else
    { insert(&(+(p)->nextPtr), val); }
}

- If insertion does not take place at the start of the list, the next list of some farther list node needs to be modified.
- This does not handle out-of-memory errors — Exercise!
Linked Lists — Deletion

void delete(CharList *p, char val) {
    if (*p == NULL || val < (*p)->data) { return; }
    else if (val == (*p)->data) { *p = freeCharListNode(*p); }
    else { delete(&((*p)->nextPtr), val); }
}

This is an example for conversion of tail-recursion into a while loop.

Linked Lists — Insertion — Recursive and Iterative

void insert(CharList *p, char val) {
    if (*p == NULL || val <= (*p)->data) {
        *p = newCharListNode(val, *p);
        insert(&((*p)->nextPtr), val);
    }
}

void insertIter(CharList *p, char val) {
    while (*p != NULL && val > (*p)->data) /* search insertion point */ {
        p = &((*p)->nextPtr);
    }
    *p = newCharListNode(val, *p);
}

This is an example for conversion of tail-recursion into a while loop.

Linked Lists — Appending

void append(CharList *p, CharList q) {
    if (*p == NULL) { *p = q; }
    else { append(&((*p)->nextPtr), q); }
}

void appendIter(CharList *p, CharList q) {
    while (*p != NULL) {
        p = &((*p)->nextPtr);
    }
    *p = q;
}

• If the first list element is deleted, the list becomes what was the rest-list after that element.
• If deletion does not take place at the start of the list, the next list of some farther list node needs to be modified.

Linked Lists — Insertion — Recursive and Iterative

void insert(CharList *p, char val) {
    CharList tmp;
    if (*p == NULL || val <= (*p)->data) {
        if (((tmp = newCharListNode(val, *p)) == NULL)
            fprintf(stderr, "insert: out of memory - not inserted.\n");
        else
            *p = tmp; /* redirect *p to inserted node */
    }
    else { insert(&((*p)->nextPtr), val); }
}

void insertIter(CharList *p, char val) {
    CharList tmp;
    while (*p != NULL && val > (*p)->data) /* search insertion point */ {
        p = &((*p)->nextPtr);
    }
    if (((tmp = newCharListNode(val, *p)) == NULL)
        fprintf(stderr, "insertIter: out of memory - not inserted.\n");
    else
        *p = tmp; /* redirect *p to inserted node */
}

• Appending at the end should be easy — we know exactly where it goes
• Nevertheless, the whole list needs to be traversed
• If the list has n elements, this means O(n) steps — linear complexity
• Adding a node at the beginning requires O(1) steps — constant complexity
Linked Lists — Iterative Deletion

typedef struct CharListNodeStruct {
    char data;
    struct CharListNodeStruct *nextPtr; /* struct label necessary! */
} CharListNode;
typedef CharListNode *CharList;

CharList freeCharListNode(CharList n) {
    CharList next = n.nextPtr;
    free(n);
    return next;
}

/* Deletion changes the argument list — pass by reference necessary! */
void deleteIter(CharList *p, char val) {
    while ( *p != NULL && val < (*p)->data ) { /* search deletion point */
        p = &((*p)->nextPtr);
    }
    if ( *p == NULL || val < (*p)->data ) { return; }
    else { *p = freeCharListNode(*p); }
}

Copying

CharList copyList_unsafe(CharList p) {
    if( p == NULL ) return NULL;
    else return newCharListNode(p->data, copyList_unsafe(p->nextPtr));
}

This version does not catch or report out-of-memory errors.

Design for a safe version:

- caller needs to be able to determine success — NULL is a legal return value for successfully copying the empty list!
- return TRUE on success FALSE on failure
- the result of copying then ends up in a pass-by-reference parameter
- on failure, take care to deallocate all memory allocated by this copying call
- on failure, don’t leave the result pointing to unfinished copies or, worse, deallocated memory!

Safe Copying

bool copyList(CharList *dest, CharList p) {
    CharList rest, head;
    if( p == NULL ) { *dest = NULL; return TRUE; }
    *dest = newCharListNode(p->data, NULL);
    if( *dest == NULL ) return FALSE;
    else {
        if ( copyList( &(*dest)->nextPtr), p->nextPtr )
            return TRUE;
        else {
            free( *dest );
            *dest = NULL;
            return FALSE;
        }
    } }

Exercise:

- Write a version that copies the rest before allocating the head.
- Write an iterative version of both
Linked Lists — Reversing

Naïve recursive solution:

- \( \text{reverse}() = \langle \rangle \)
- \( \text{reverse}(\langle x_1 \rangle + \langle x_2, \ldots, x_n \rangle) = \text{reverse}(\langle x_2, \ldots, x_n \rangle) + \langle x_1 \rangle \)

```c
void reverse(CharList *p) {
    CharList tmp = *p;
    if (*p == NULL) { return; }
    else { *p = tmp->nextPtr; 
              tmp->nextPtr = NULL; 
              reverse(p); 
              append(p, tmp); }
}
```

- \( \text{append}(p1, p2) \) needs time linear in the length of list \( p1 \)
- \( \text{reverse}(p) \) calls \( \text{append} \) \( n \) times, with average first-argument length \( \frac{n}{2} \)
- \( \text{reverse}(p) \) therefore needs time \textbf{quadratic} in the length of list \( p — O(n^2) \)

Linked Lists — Linear-time Iterative Reversing

```c
void reserverter(CharList *p) { /* linear complexity */
    CharList prev = NULL, current = *p, next;
    while (next != NULL) {
        *p = next;
        prev = current;
        current = next;
        next = current->nextPtr;
        current->nextPtr = prev;
    }
}
```

- Sequence reversing starts from beginning of list
- Time: \( O(n) \), i.e., linear in the length of the argument list
- Space: \( O(1) \): one constant-size stack frame containing \( p, \text{prev}, \text{current}, \text{next} \)

Linked Lists — Linear-time Recursive Reversing

```c
void reverse(CharList *p) { if (*p != NULL) *p = rev(*p); } 
```

Recursive auxiliary function \( \text{rev} \):
- Precondition: \( q \neq NULL \)
- Returns pointer to original last node
- Original first node stays in place — caller can still access it

```c
CharList rev(CharList q) {
    CharList next = q->nextPtr;
    if (next == NULL) return q;
    else {
        CharList result = rev(next);
        next->nextPtr = q;
        q->nextPtr = NULL;
        return result;
    }
}
```

- Sequence reversing starts from end of list
- Time: \( O(n) \), i.e., linear in the length of the argument list
- Space: \( O(n) \): one stack frame per list element