ECE220 Lab7

Brain Teaser – Linked List In-order insertion

```c
// Single pointer insertion
void insert_list_single(node_t **list, int val)
{
    node_t *cur = *list, *prev = NULL;

    // Search for location
    while (cur != NULL && cur->val <= val)
    {
        prev = cur;
        cur = cur->next;
    }

    // Insert node at beginning of list
    if (prev == NULL)
    {
        node_t *temp = (node_t*)malloc(sizeof(node_t));
        temp->val = val;
        temp->next = cur;
        *list = temp;
    }

    // Insert node in middle or end of list
    else
    {
        node_t *temp = (node_t*)malloc(sizeof(node_t));
        temp->val = val;
        temp->next = cur;
        prev->next = temp;
    }
}
```

Design an algorithm to insert elements into a linked list in-order. For example, given the list and element 8, an in-order insertion would look like:

Before: $1 \rightarrow 3 \rightarrow 9$

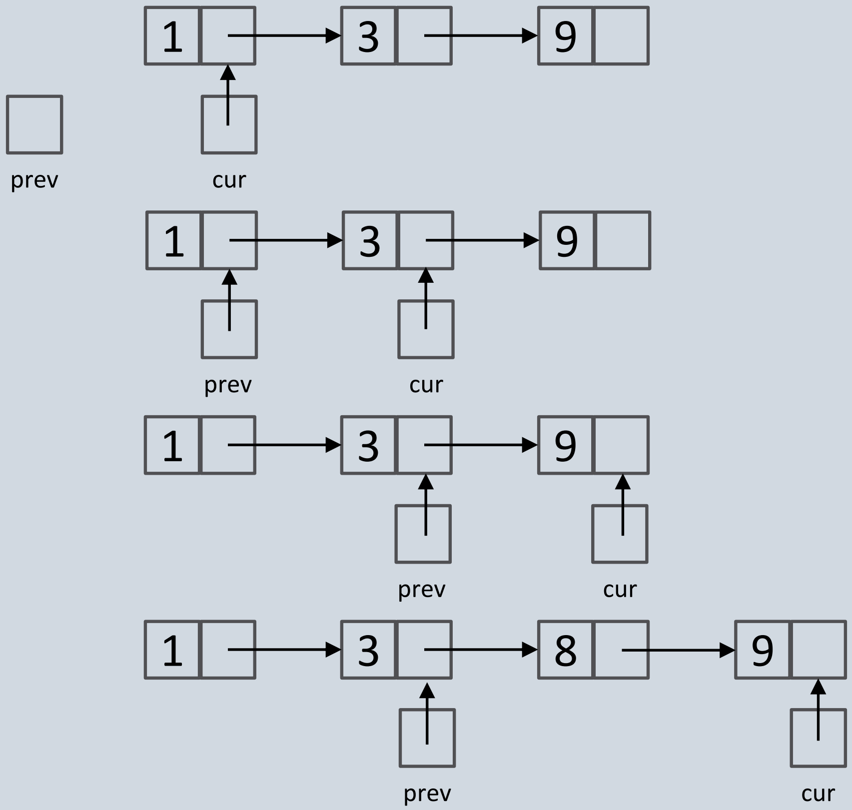
After: $1 \rightarrow 3 \rightarrow 8 \rightarrow 9$

Solution

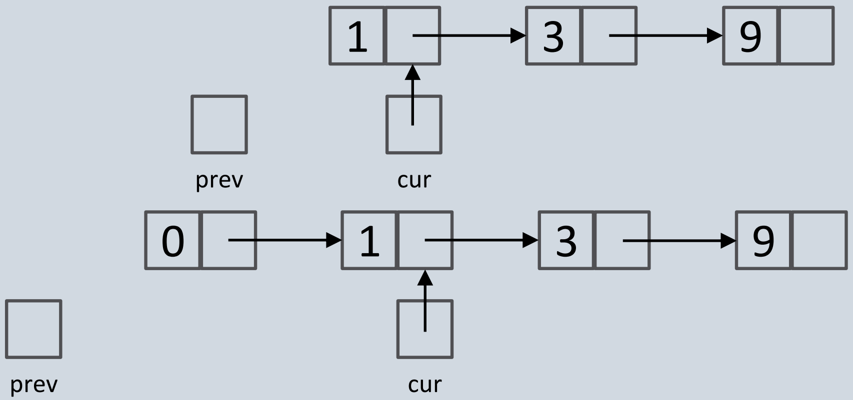
- Use two pointers, *prev* and *cur*, to keep track of the previous node and the current node.
- Iterate *cur* until it reaches the end of the list or find place to insert
- Insert element into the list but handle case where insertion happens at the very beginning.

Brain Teaser – Single Pointer Insertion

INSERTING 8



INSERTING 0



Brain Teaser – Linked List In-Order Insertion with Double Pointers

```c
// Double pointer insertion
void insert_list_double(node_t **list, int val)
{
    node_t **cur = list;
    while (*cur != NULL && (*cur)->val < val)
    {
        cur = &((*cur)->next);
    }

    node_t *temp = (node_t*)malloc(sizeof(node_t));
    temp->val = val;
    temp->next = (*cur);
    (*cur) = temp;
}
```

Use a double pointer to do insertion rather than single pointer

Advantages

- Only needs a single 'pointer'
- Less logic and can handle any insertion point in the list

Disadvantages

- Traversing and accessing list requires more thought

List insertion – Side by Side Comparison

```c
// Single pointer insertion
void insert_list_single(node_t **list, int val)
{
    node_t *cur = *list, *prev = NULL;

    // Search for location
    while (cur != NULL && cur->val <= val)
    {
        prev = cur;
        cur = cur->next;
    }

    // Insert node at beginning of list
    if (prev == NULL)
    {
        node_t *temp = (node_t*)malloc(sizeof(node_t));
        temp->val = val;
        temp->next = cur;
        *list = temp;
    }

    // Insert node in middle or end of list
    else
    {
        node_t *temp = (node_t*)malloc(sizeof(node_t));
        temp->val = val;
        temp->next = cur;
        prev->next = temp;
    }
}
```

```c
// Double pointer insertion
void insert_list_double(node_t **list, int val)
{
    node_t **cur = list;
    while (*cur != NULL && (*cur)->val < val)
    {
        cur = &((*cur)->next);
    }

    node_t *temp = (node_t*)malloc(sizeof(node_t));
    temp->val = val;
    temp->next = (*cur);
    (*cur) = temp;
}
```

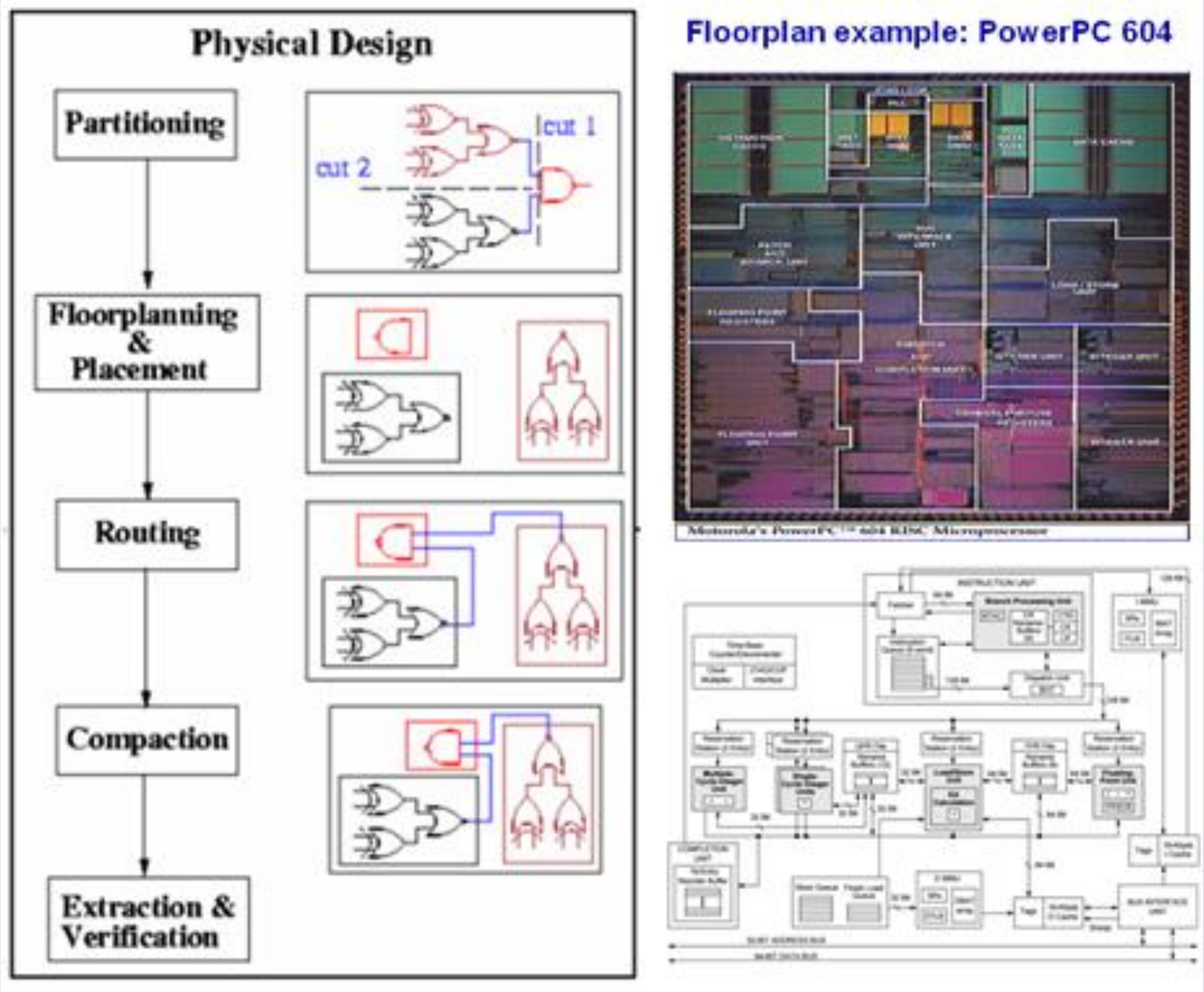
Brain Teaser – Freeing a list

```c
void destroy_list(node_t *list)
{
    for (node_t *cur = list; cur != NULL;)
    {
        node_t *temp = cur;
        cur = cur->next;
        free(temp);
    }
}
```

How to completely free a list?

Solution

- Iterate through each node in list
- Free each node individually but need to watch out when freeing so that can still access later elements

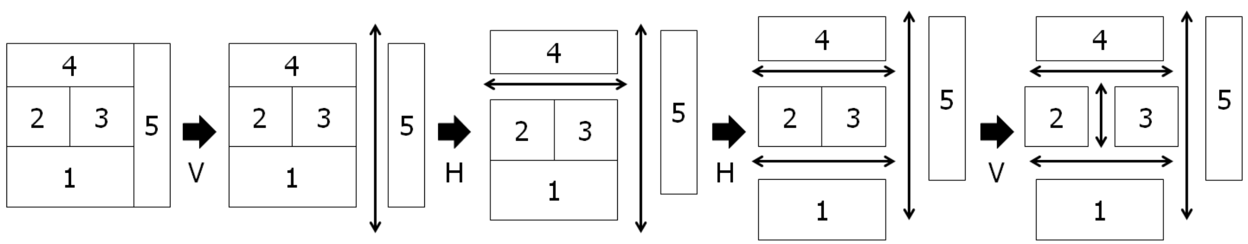


MP7 – Optimizing Floorplans

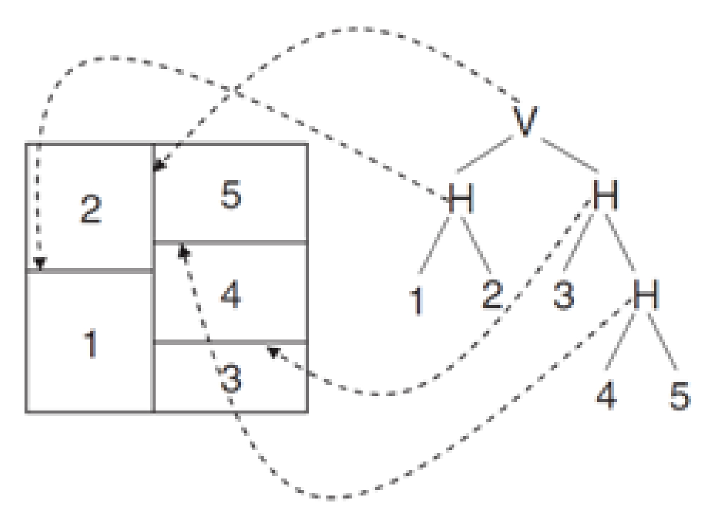
Floorplanning is the process of assembling partitioned circuit modules in an optimal fashion to maximize some metric.

Implement a floorplanner to layout module and optimize packing area.

MP7 – Floorplan Models



A slicing floorplan. Each rectangular macro can be cut out vertically or horizontally.

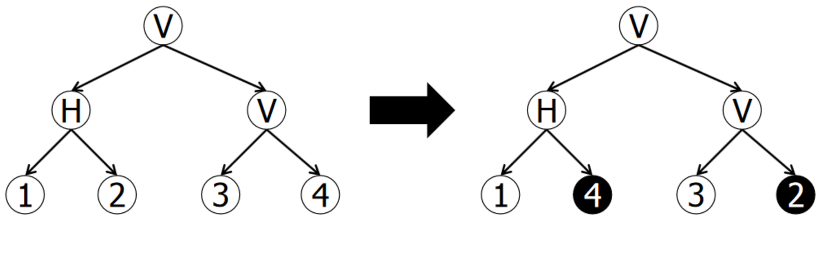


Modules in a floor plan can be cut horizontally or vertically.

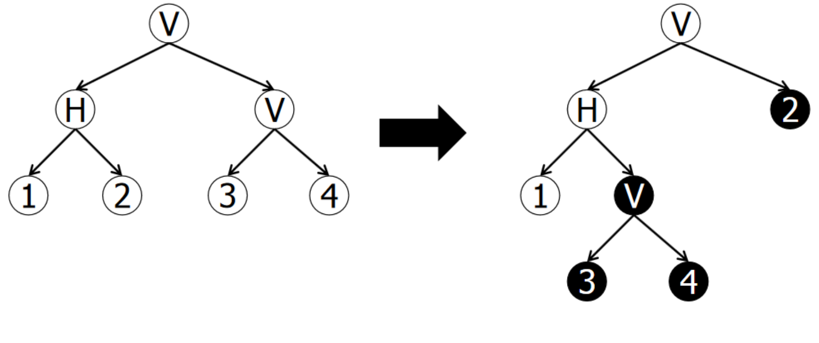
Floorplans can be represented as trees and have a corresponding postfix expression

Postfix expression for lower image is

$$12H345HHV$$



Operation: swap_module



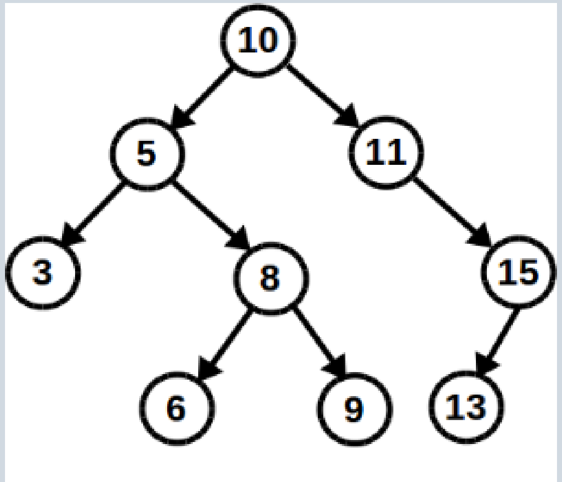
Operation: swap_topology

MP7 – Floorplan Functions

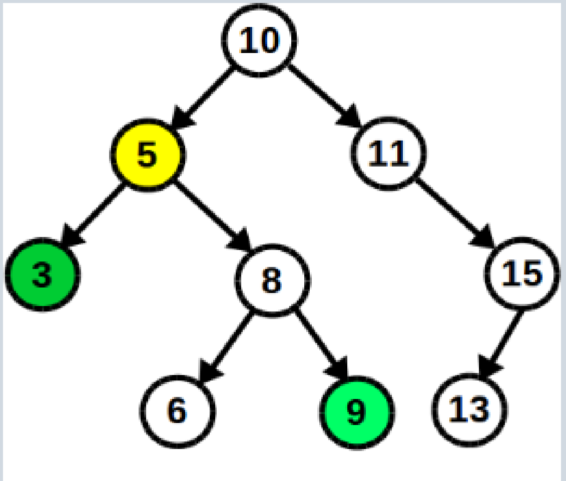
Functions to implement

- *init_slicing_tree* – generate an initial slicing tree
- *get_expression* – perform a postfix traversal on the slicing tree and extract expression
- *recut* – change the cutline of an internal node
- *rotate* – swap the height and width of a module
- *swap_module* – swap two modules from two leave nodes (do this by swapping pointer values rather than actual pointers)
- *swap_topology* – swap two subtrees rooted at two given node pointers by modifying the node links appropriately

Lab7 – Lowest Common Ancestor



A binary search tree



The LCA of 3 and 9 is 5

Given a binary search tree (BST), find the lowest common ancestor of two nodes.

Hint: use the special property of a (BST) to figure out what the lowest common ancestor is. If this was a general tree, then you would have to do a recursive search to find the lowest common ancestor.

MP8 – C++ Calculator

Implement a simple calculator in C++

Supports
- Real numbers - $r \in \mathbb{R}$
- Complex numbers - $(\mathrm{a} + \mathrm{bi}) \in \mathbb{C}$
- Rational numbers - $\frac{p}{q} \in \mathbb{Q}$

Implement each number type as a child of *Number*
- support four operators - $+,\ -,\ \times,\ /$
- functions
  - *magnitude*
  - *print*
  - *set_value*

Implement constructor and overloaded functions for *ComplexNumber* and *RationalNumber*

Lab8 – C++ Classes

Familiarize with various C++ concepts and implement them

- Constructor (and destructor)
- Setter and getter functions
- Operational overloads

Implement methods of Rectangle class

Utilize implementation previous implementation to perform computations on a list of Rectangles.