Lecture 4: Stack, Queue, and VLSI Floorplan

Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Recap Sorting

- Complexity of popular sorting algorithms

Bubble Sort	$O(n^2)$
Insert Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Quick Sort	**Average** $O(n \log n)$
Merge Sort	$O(n \log n)$
Heap Sort	$O(n \log n)$
Radix Sort	$O(n \log n)$

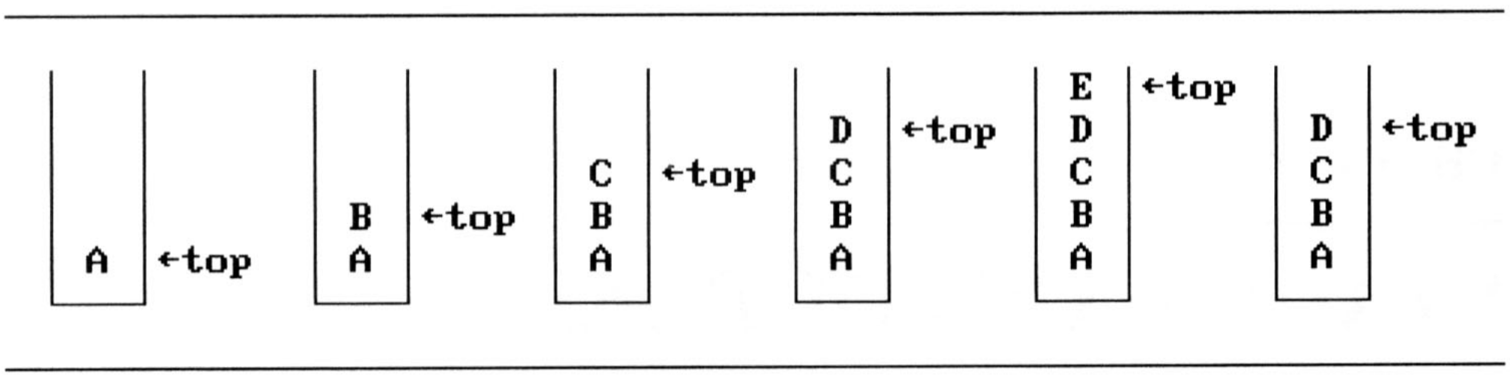
In fact, what you need may be …

- Use std::sort in 90% problems
  - https://en.cppreference.com/w/cpp/algorithm/sort

```
std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

// sort using the default operator<
std::sort(s.begin(), s.end());
for (auto a : s) {
    std::cout << a << " ";
}
std::cout << '\n';

// sort using a standard library compare function object
std::sort(s.begin(), s.end(), std::greater<int>());
for (auto a : s) {
    std::cout << a << " ";
}
std::cout << '\n';
```

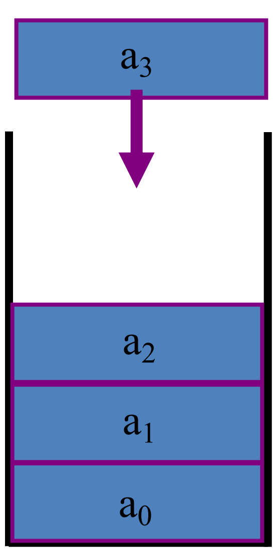
Stack

- A stack is an ordered list in which insertions and deletions are made at one end called the *top*.
  - Support **push** and **pop** operations and **top** query
- If we add the elements *A*, *B*, *C*, *D*, *E* to the stack, in that order, then *E* is the first element we delete from the stack
- A stack is also known as a *Last-In-First-Out* (*LIFO*) list.

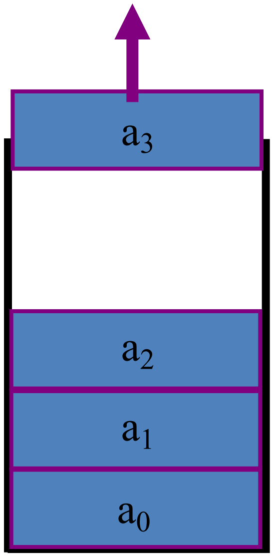


Visualization of Stack

- **Main Subroutine**
  - Push
  - Pop
  - Top



Push (Add)



Pop (Delete)

std::stack

- C++ Standard Template Library (STL) stack
  - https://en.cppreference.com/w/cpp/container/stack

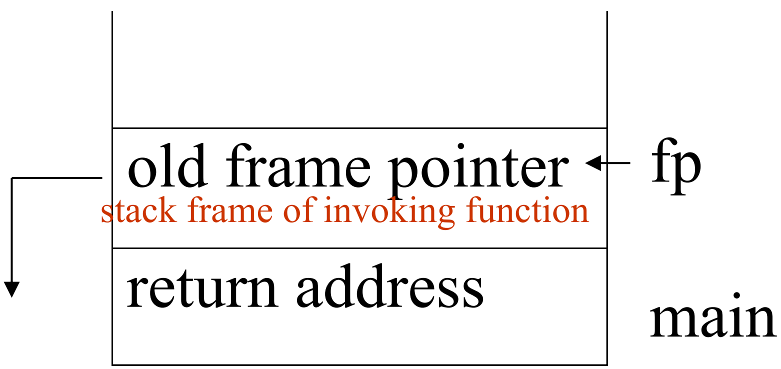
```
/* stack example */
#include <iostream>
#include <stack>

int main()
{
    std::stack<int> stk;
    stk.push(1);
    stk.push(2);
    std::cout<<stk.top();

    /* clear the stack */
    while(!stk.empty())
      stk.pop();
}
```

Application: Recursion Stack Frame

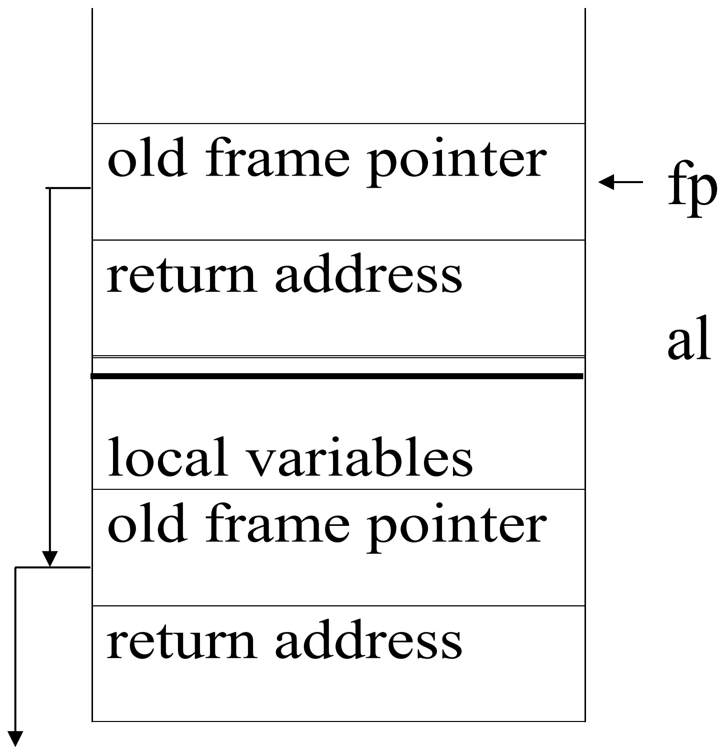
stack frame of recursive function call

(activation record)

fp: a pointer to current stack frame



system stack before a1 is invoked

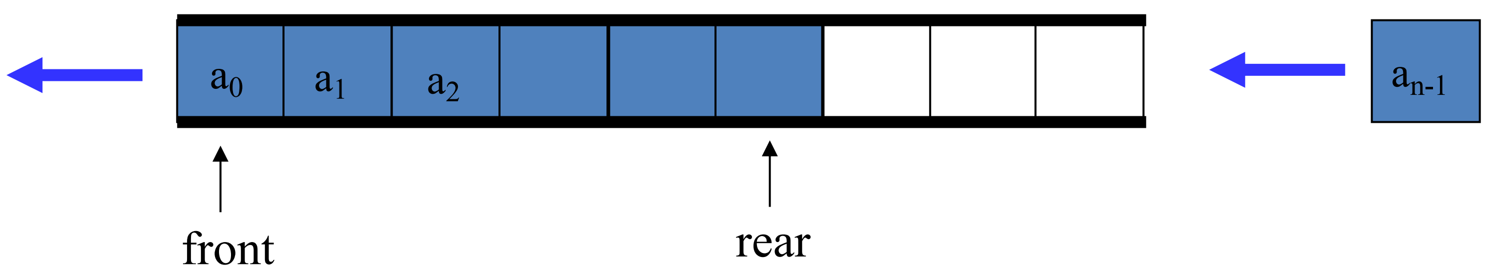


system stack after a1 is invoked

All recursive algorithm can be rewritten iteratively using either flat for-loop or stack

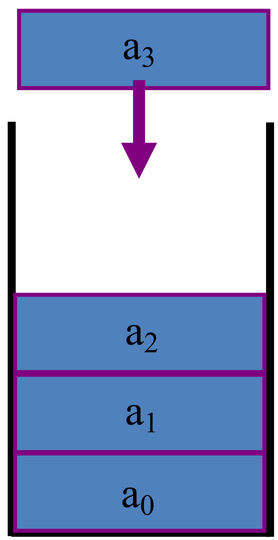
Queue

- A queue is an ordered list in which insertions and deletions are made at one end called the rear and front
  - Support **push** and **pop** operations and **front** query
- If we add the elements *A*, *B*, *C*, *D*, *E* to the stack, in that order, then *A* is the first element we delete from the queue
- A queue is also known as a *First-In-First-Out* (*LIFO*) list

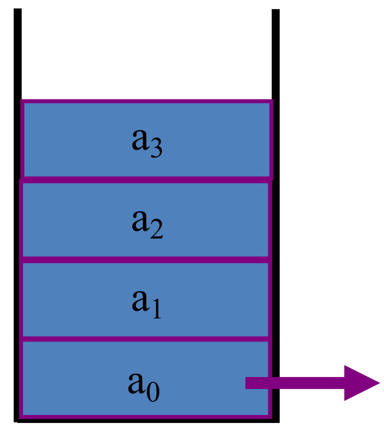


Visualization of Queue

- **Main Subroutine**
  - Push
  - Pop
  - front



Push (Add)



Pop (Delete)

std::queue

- C++ Standard Template Library (STL) queue
  - https://en.cppreference.com/w/cpp/container/queue

```
#include <iostream>
#include <queue>
int main()
{
    std::queue<int> que;
    que.push(1);
    que.push(2);
    std::cout<<que.front();

    /* clear the stack */
    while(!que.empty())
      que.pop();
}
```

Example: Parenthesis Problem

- Given a string of characters '(', ')', '{', '}', '[' and ’]’
- Goal: Determine if the input string is valid.
  - An input string is valid if:
  - Open brackets must be closed by the same type of brackets.
  - Open brackets must be closed in the correct order.
  - Note that an empty string is also considered valid.

()	valid
()[]{}	valid
(]	invalid
([)]	invalid
{[]}	valid

Applications

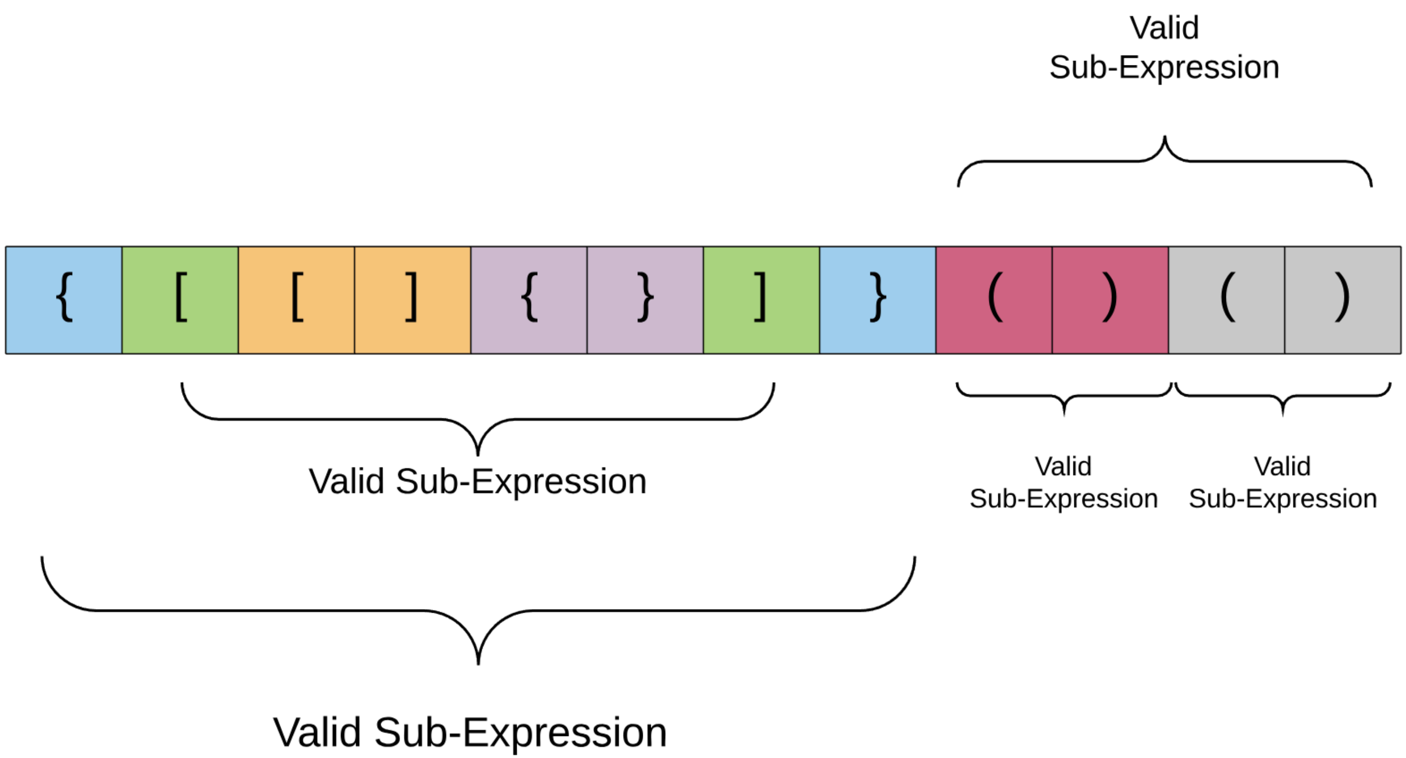
- **A fundamental routine in language compiler**
  - Need to parse a valid mathematical expressions
    - (3+2)*4*((9-6)/6)
    - (double)(1)/(2+7)
  - Need to parse a valid code block
    - int main () { return 0; }
    - void function() {}
    - auto lambda = [] () { my_work(); }
- **Also a very frequently asked question in interview …**

So, by how?

()	valid
()[]{}	valid
(]	invalid
([)]	invalid
{[]}	valid
(((((())))))	valid
()()()()	valid
(((((((()	invalid
((()(())))	

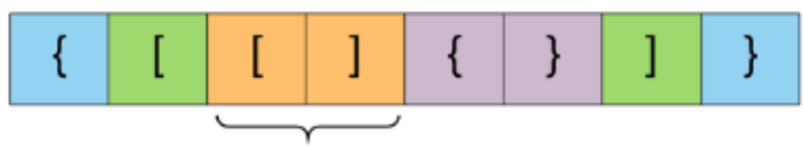
Property

- **A valid expression must imply:**
  - All subexpressions are valid



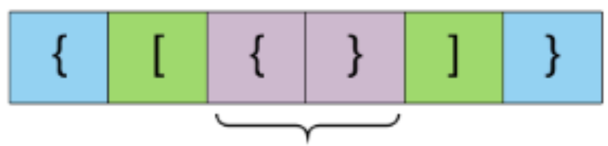
Recursive Validness

- Remove yellow pair



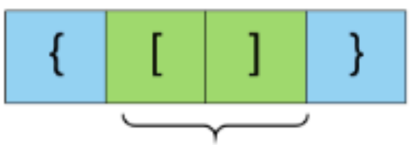
Recursive Validness

- **Remove purple pair**



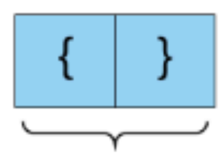
Recursive Validness

- Remove green pair



Recursive Validness

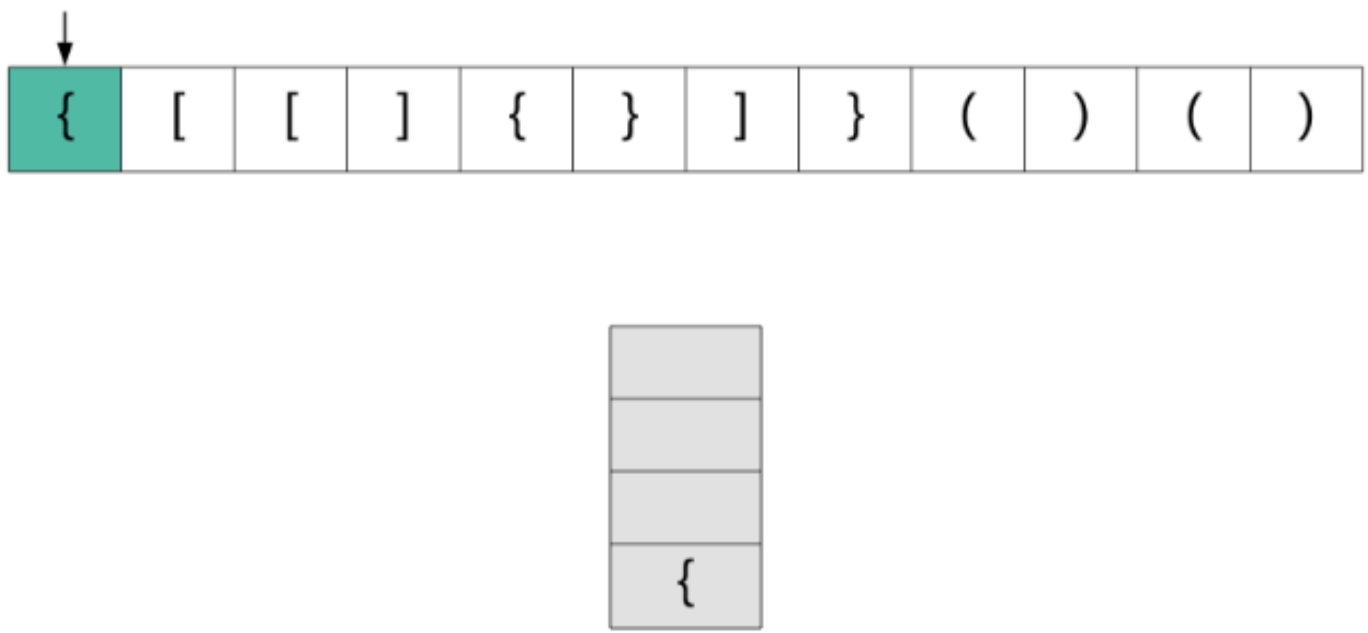
- Every subexpression is valid



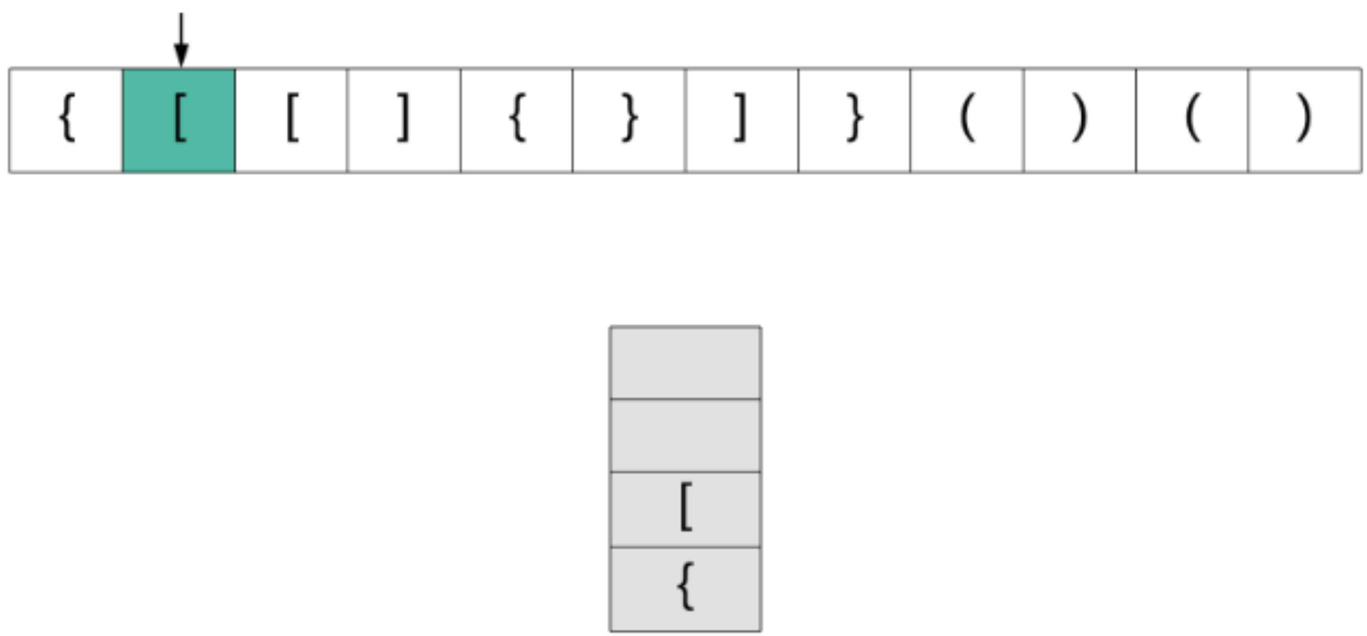
Algorithm

1. Initialize a stack S.
2. Process each bracket of the expression one at a time.
3. If we encounter an opening bracket, we simply push it onto the stack. This means we will process it later, let us simply move onto the **sub-expression** ahead.
4. If we encounter a closing bracket, then we check the element on top of the stack. If the element at the top of the stack is an opening bracket of the same type, then we pop it off the stack and continue processing. Else, this implies an invalid expression.
5. In the end, if we are left with a stack still having elements, then this implies an invalid expression.

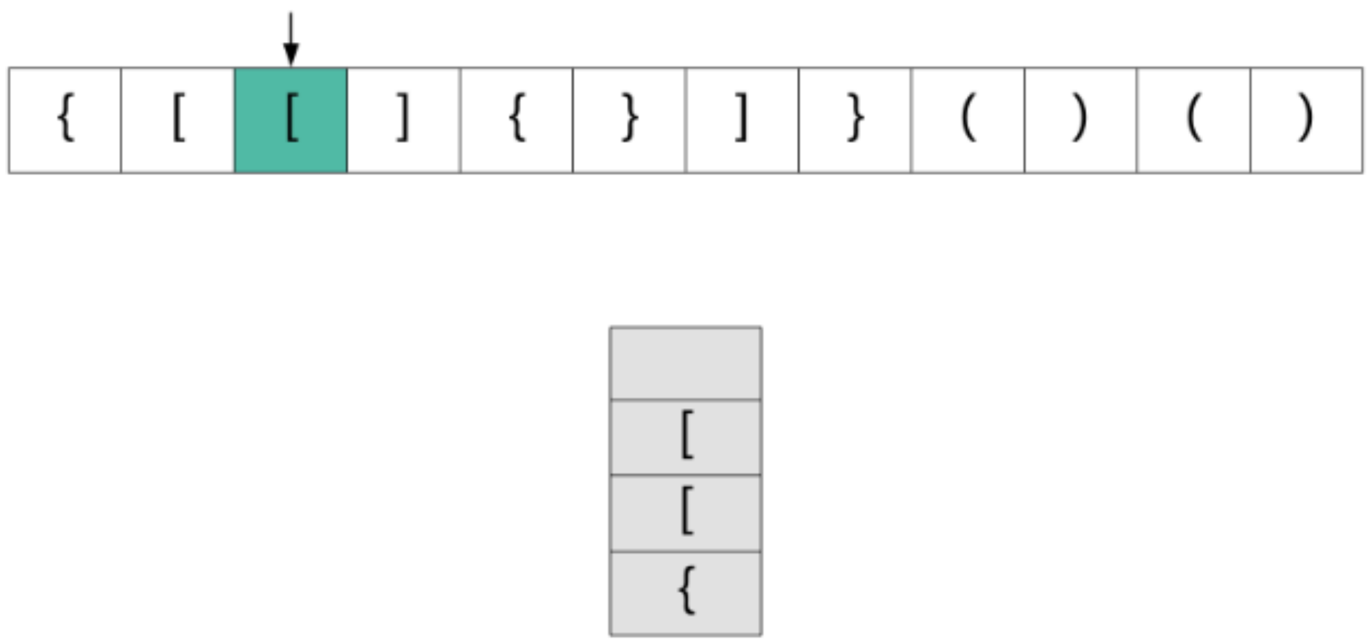
Illustration



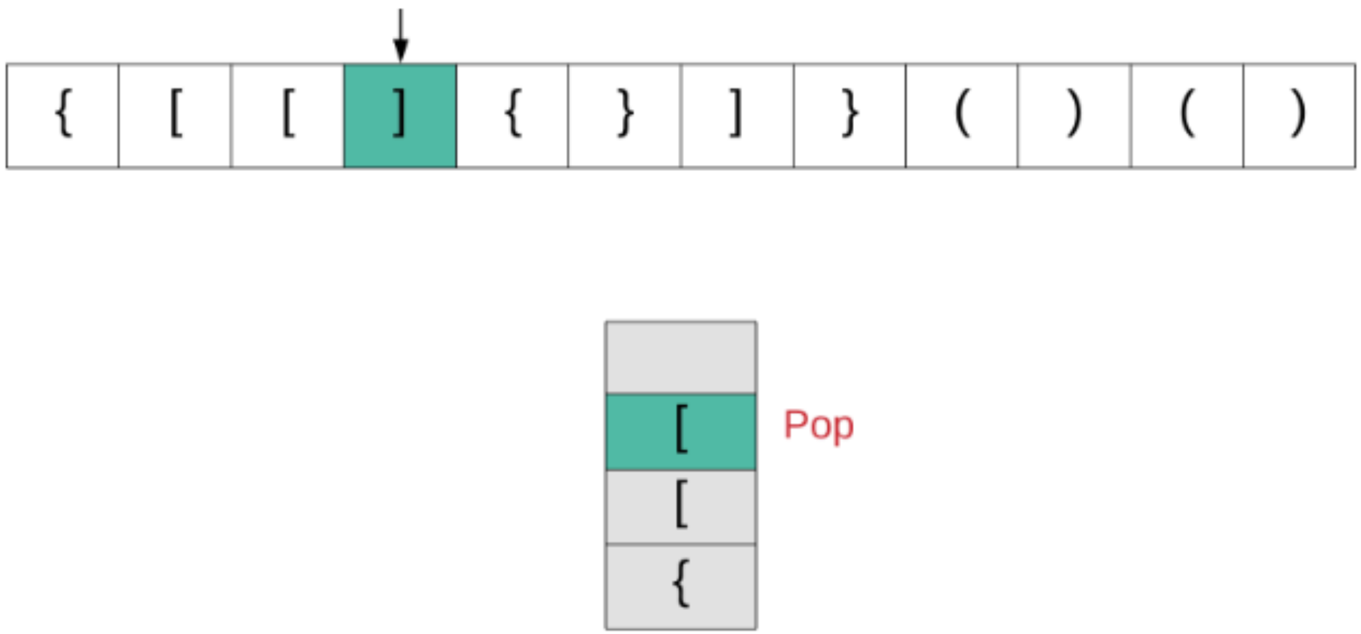
Illustration



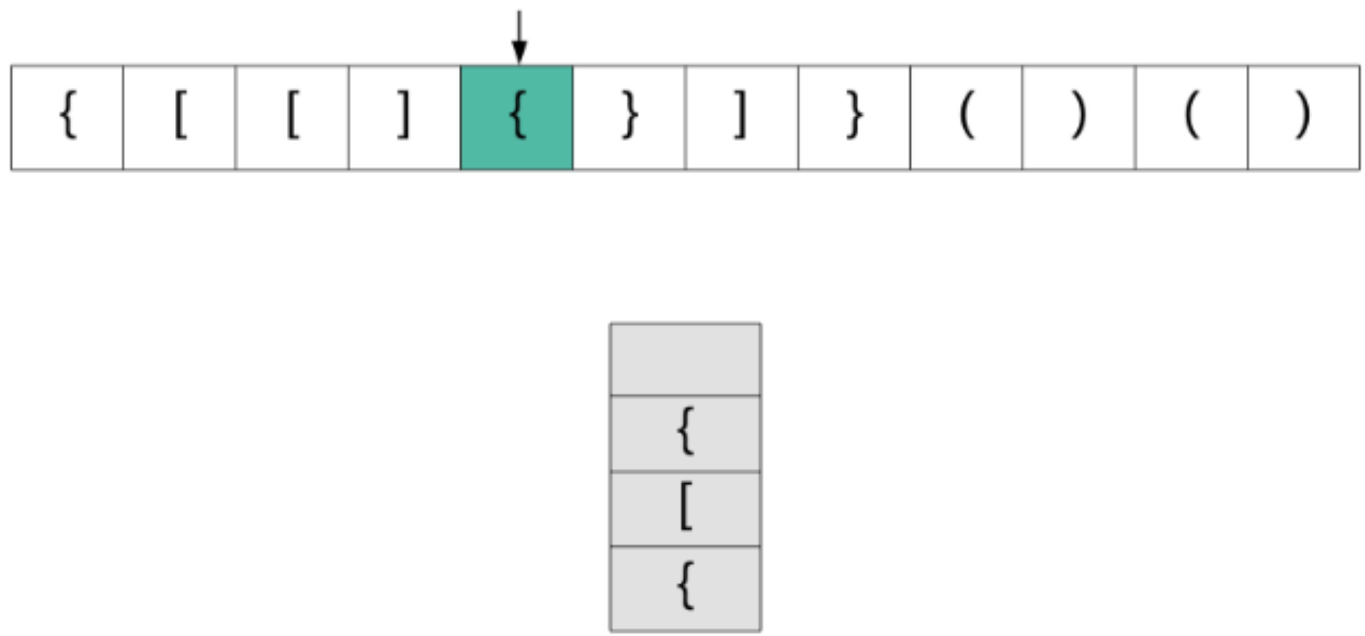
Illustration



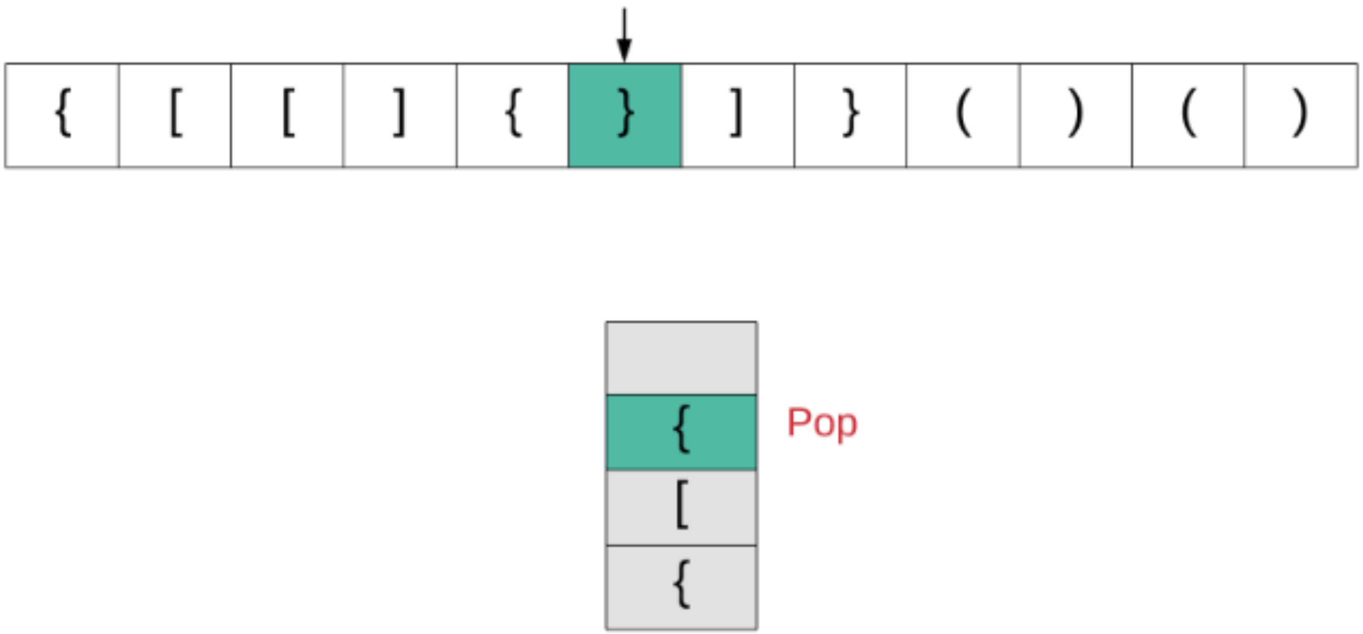
Illustration



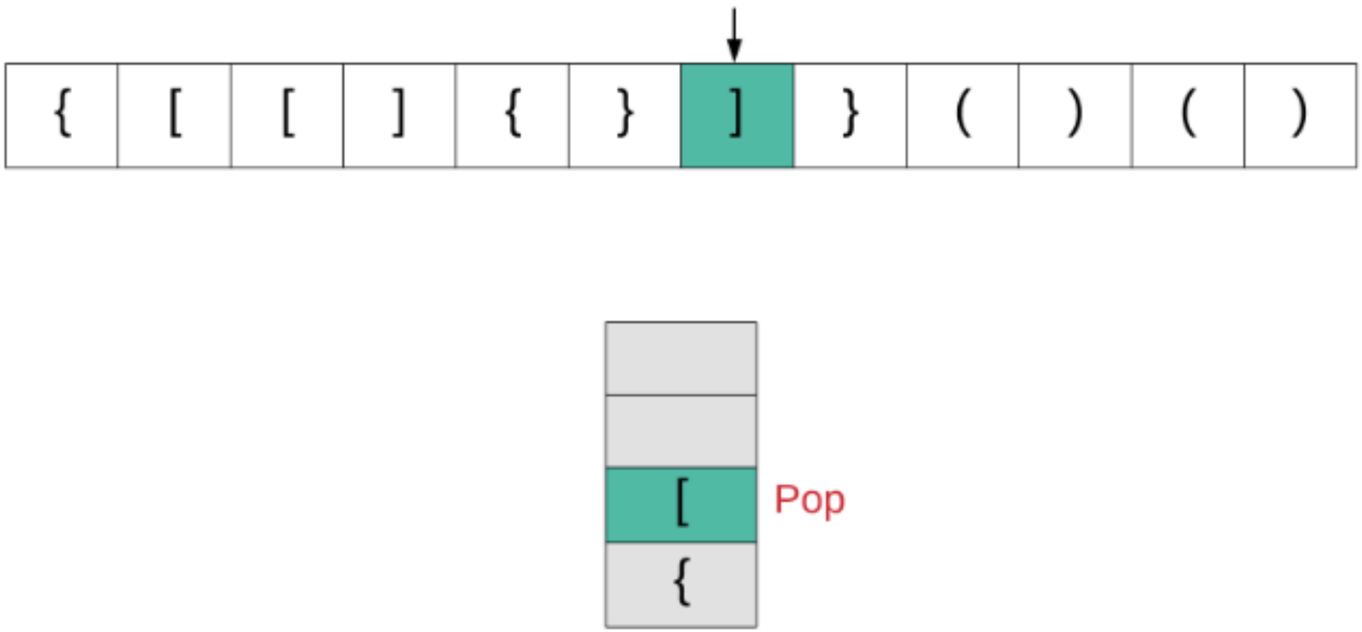
Illustration



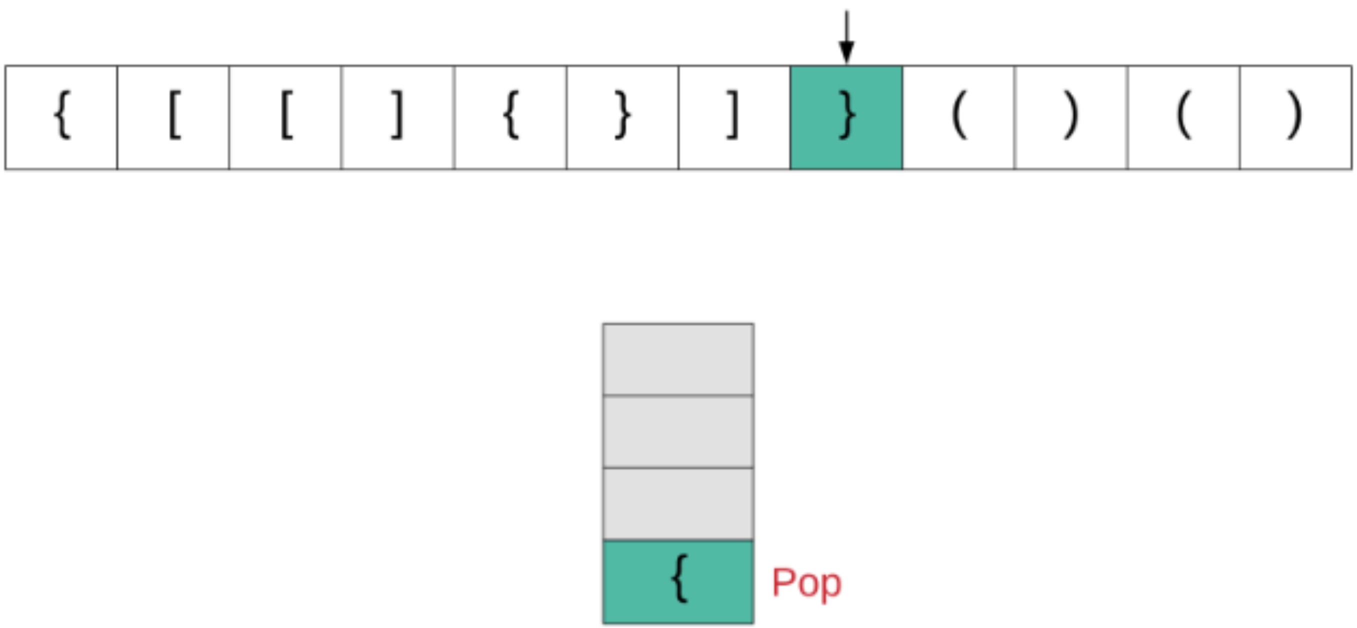
Illustration



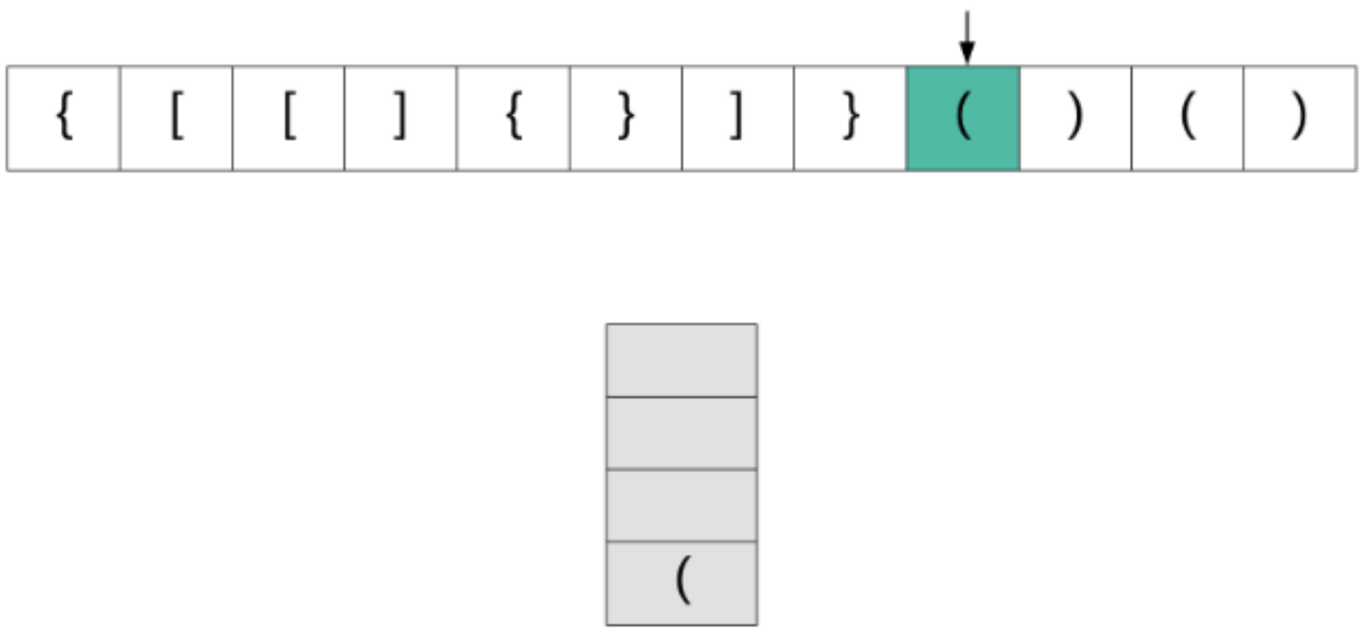
Illustration



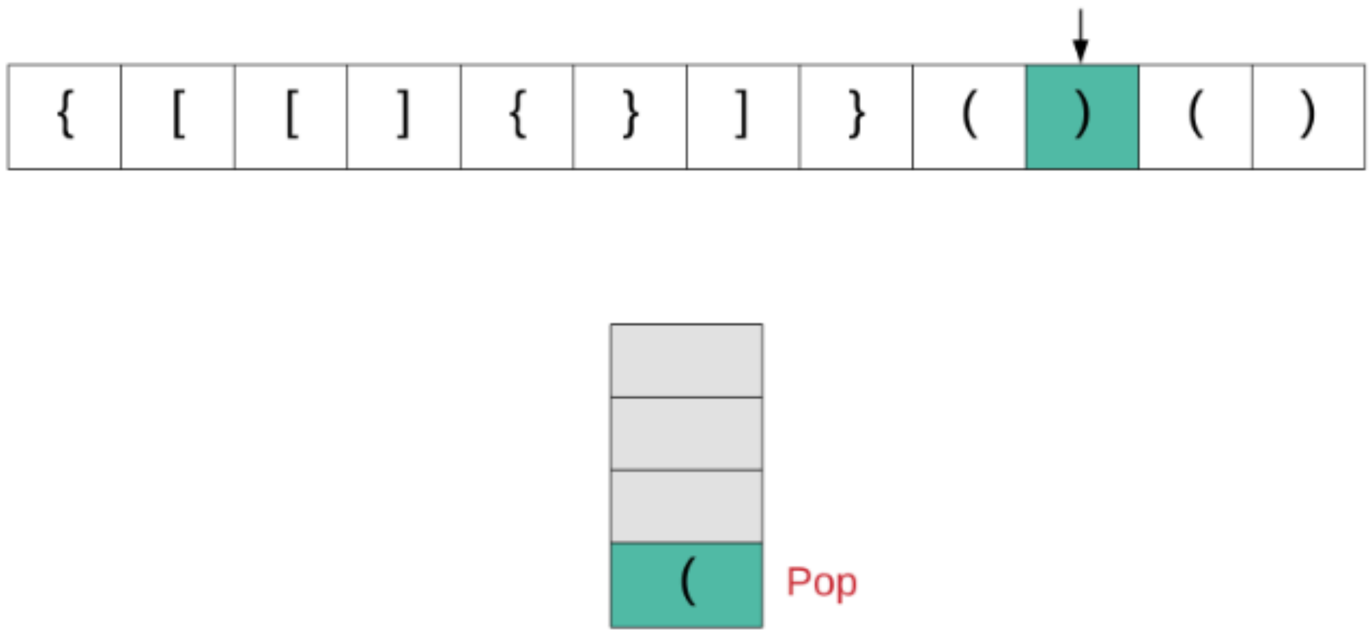
Illustration



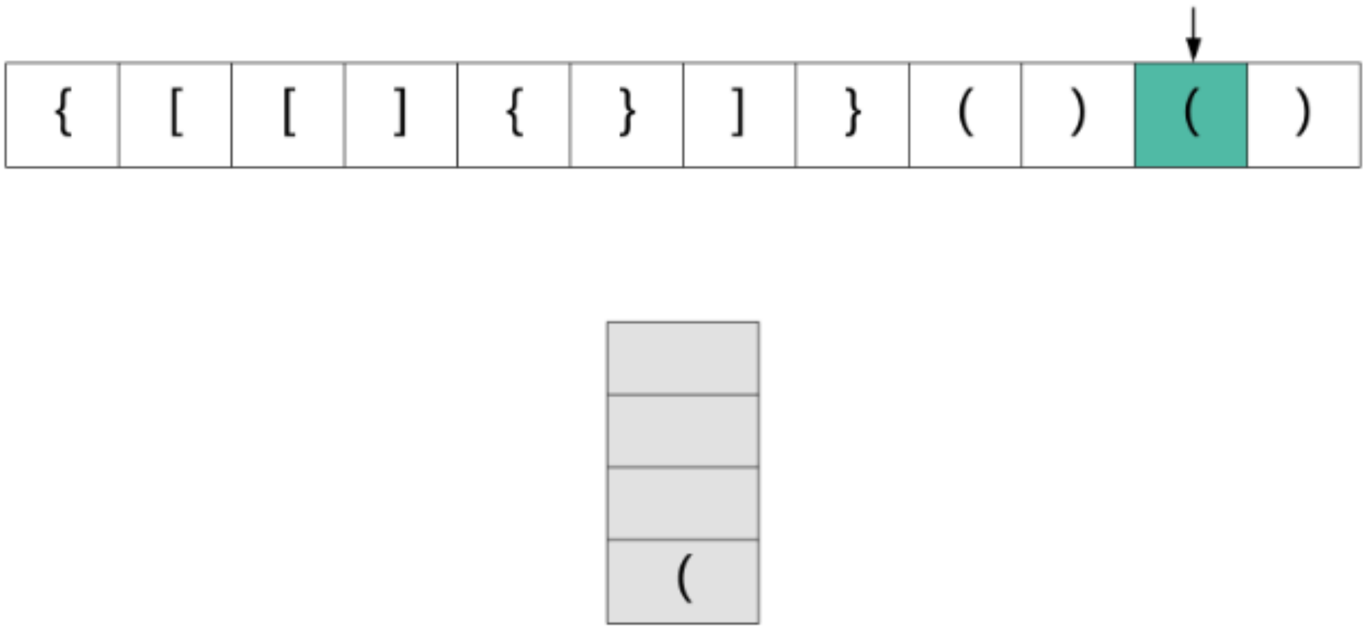
Illustration



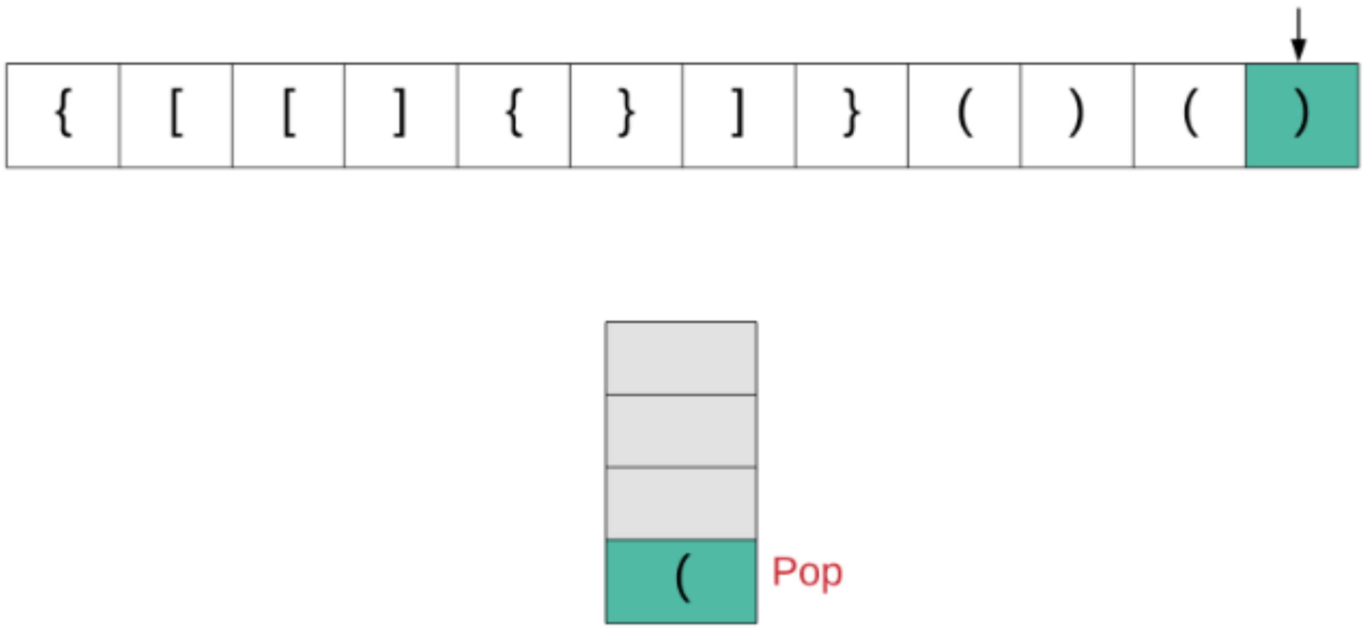
Illustration



Illustration



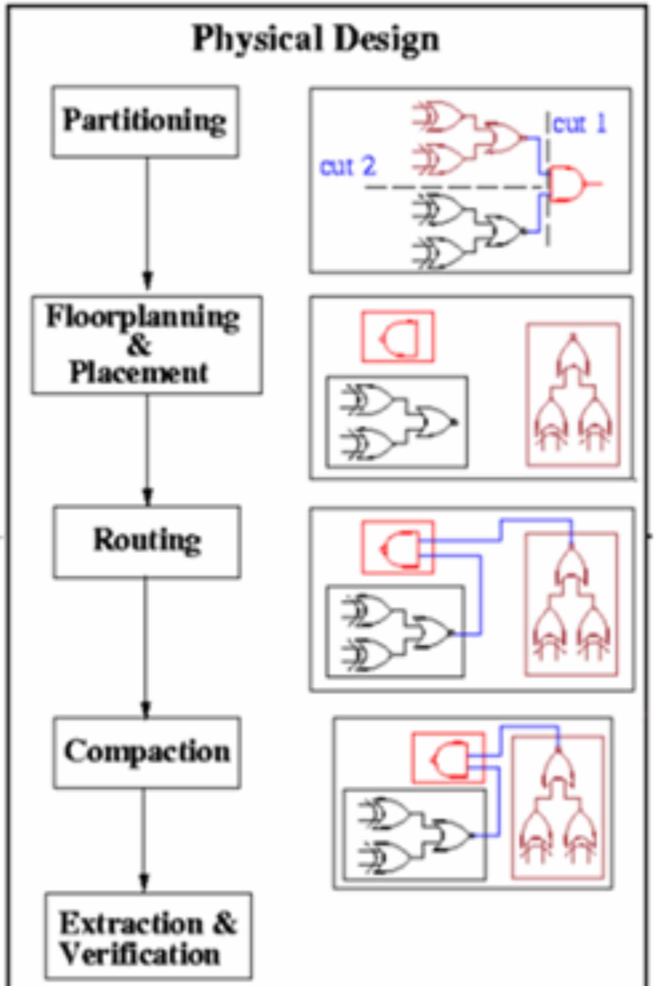
Illustration



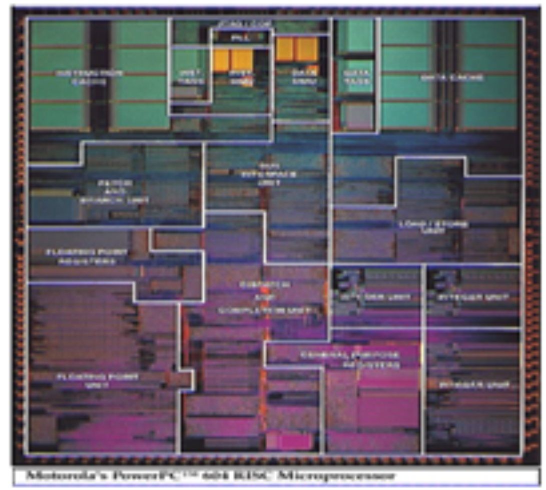
Complexity

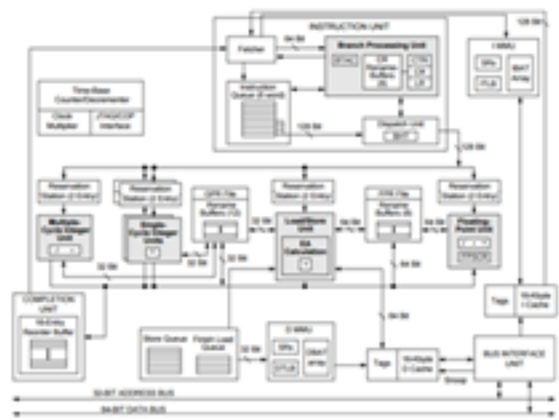
- Time complexity : $O(n)$
  - We traverse the given string one character at a time and push and pop operations on a stack take $O(1)$ time.
- Space complexity : O(n)
  - We push all opening brackets onto the stack and in the worst case, we will end up pushing all the brackets onto the stack. e.g. ((((((((((.

Design Automation Flow of IC



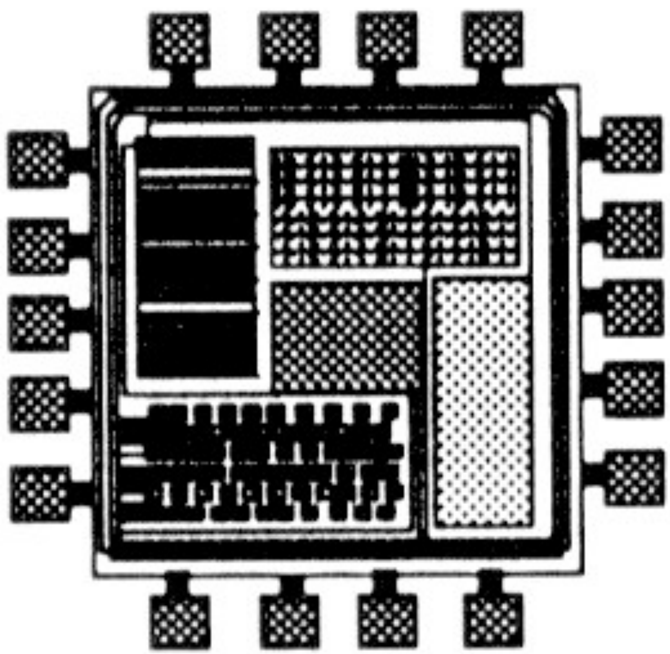
Floorplan example: PowerPC 604





Physical Layout Representation Model

- Several blocks after partitioning
  - Need to put these blocks together



Floorplanning

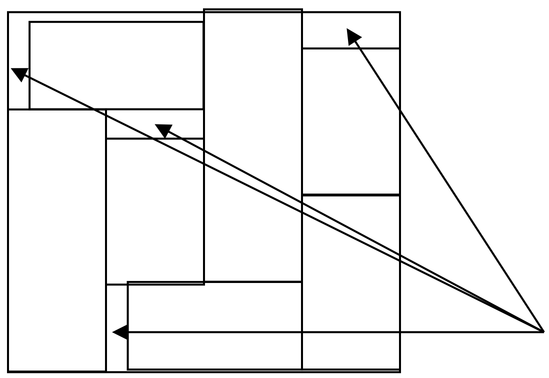
- The floorplanning problem is to plan the positions and shapes of the modules at the beginning of the design cycle to optimize the circuit performance…
  - chip area
  - total wirelength
  - delay of critical path
  - routability
  - others, ex: noise, heat dissipation, …

Floorplanning Problem Formulation

- **Input:**
  - *n* Blocks with areas $A_1, \ldots, A_n$
  - Each block has a fixed width and height
- **Output:**
  - Coordinates (x_i, y_i), width w_i and height h_i for each block
- **Constraints:**
  - Blocks cannot overlap with each other
- **Objective:**
  - To optimize the circuit performance

Avoid Dead Space

- Dead space is the space that is wasted:

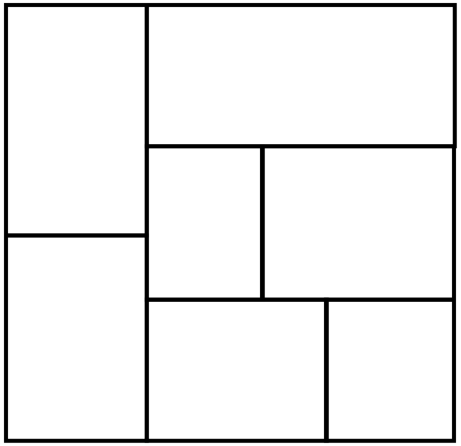


- Minimizing area is the same as minimizing dead space
- Dead space percentage is computed as

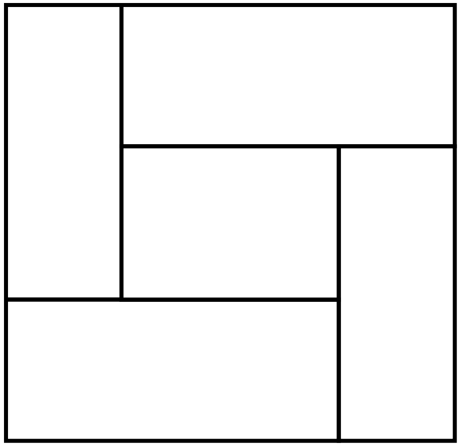
$$(A - \Sigma_i A_i) / A \times 100\%$$

Computational Representation

- **Slicing Floorplan:**
  - One that can be obtained by repetitively subdividing (slicing) rectangles horizontally or vertically

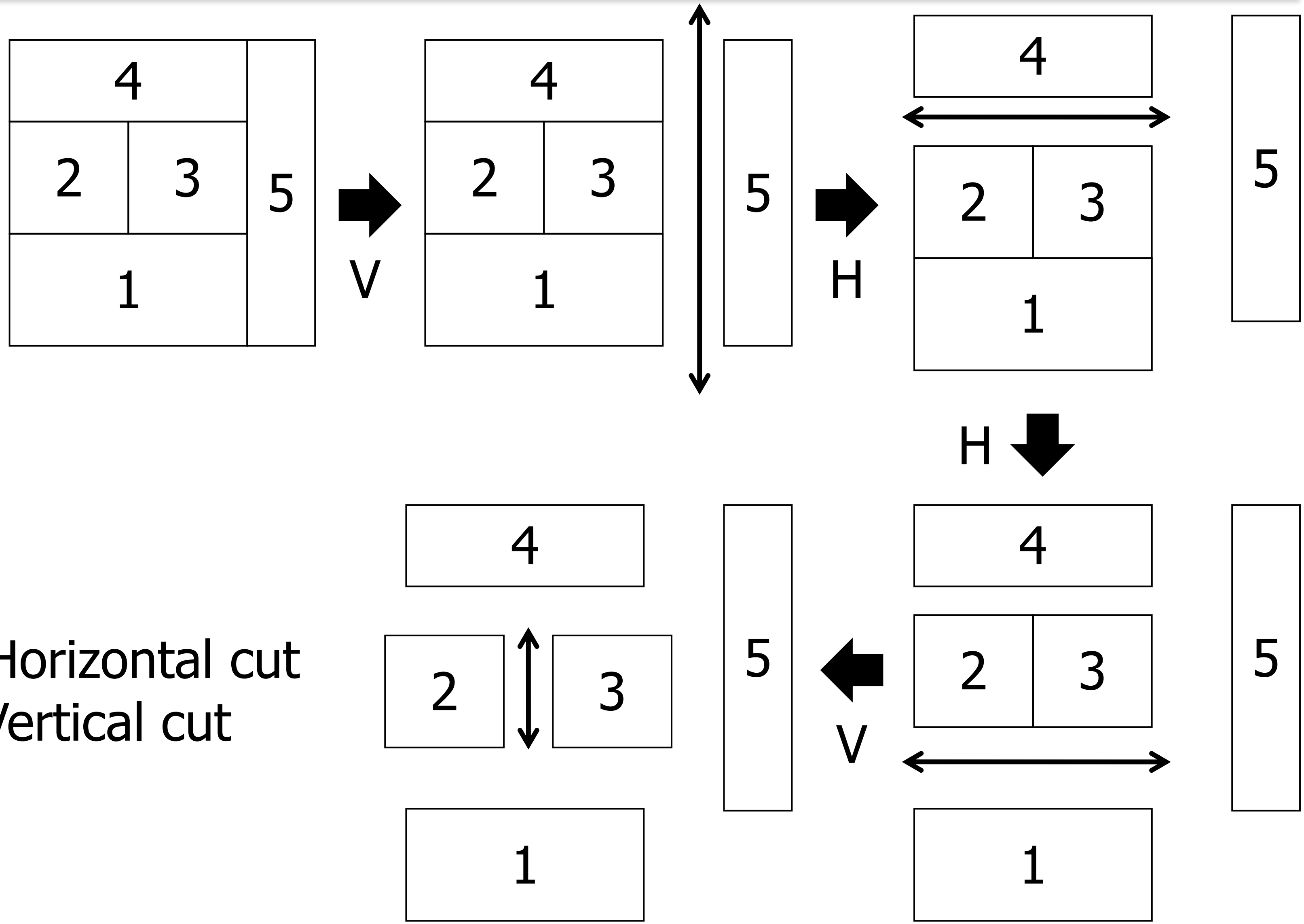


- **Non-Slicing Floorplan:**
  - One that may not be obtained by repetitively subdividing alone



- **Apparently, slicing floorplans are much easier to handle**

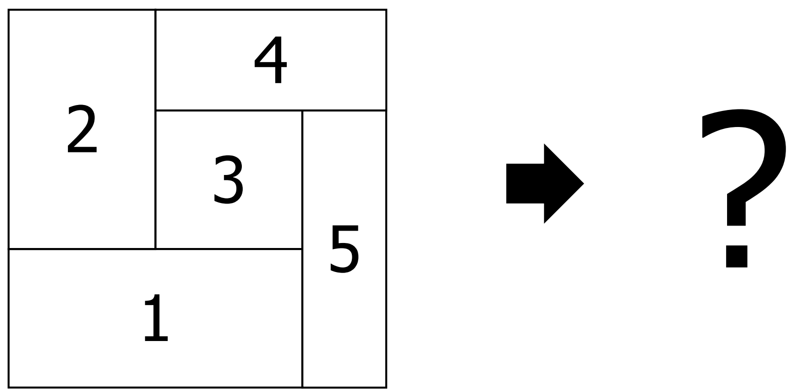
Slicing Floorplan Example



H: Horizontal cut
V: Vertical cut

Non-slicing Floorplan Example

- How do you cut this example through V and H?



A non-slicing floorplan.

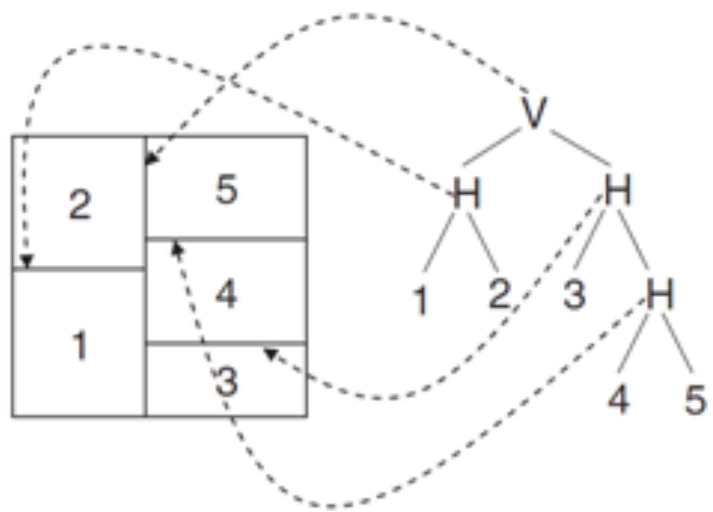
Classic Work on Floorplanning

Simulated Annealing using Polish Expression Representation

D.F. Wong and C.L. Liu,
"A New Algorithm for Floorplan Design"
DAC, 1986, pages 101-107.

Slicing Tree Representation

- A binary tree (complete)
- Modules on leave nodes & Cutlines on internal nodes
- 1D expression by postfix traversal

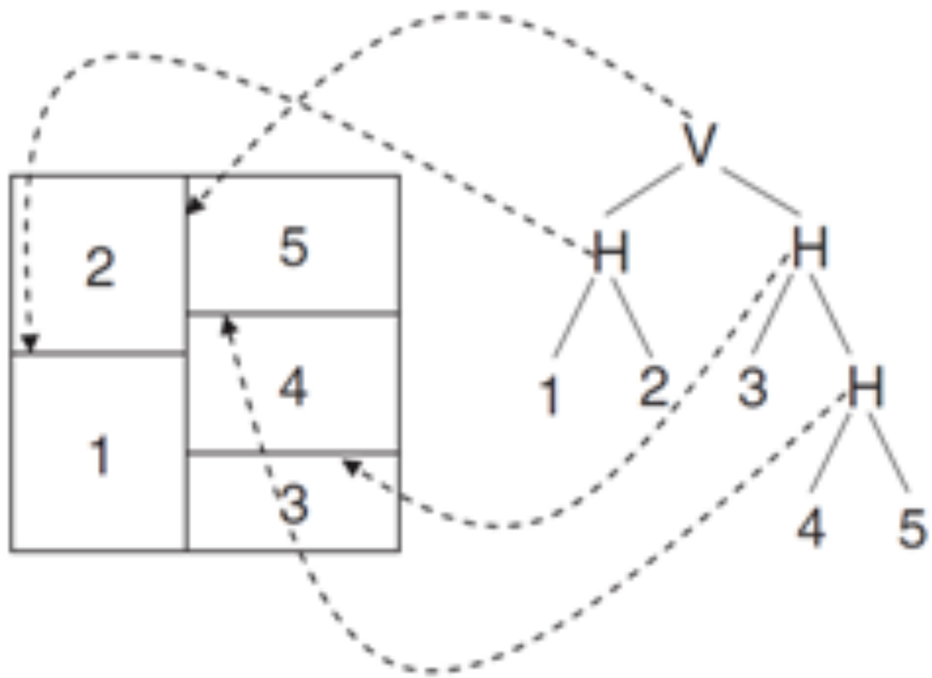


Postfix Traversal on a Binary Tree

```
Void postfix(Node* node) {
    if(node == nullptr) return;
    postfix(node->left_node);
    postfix(node->right_node);
    std::cout << node->id << '\n';
}
```

Example

- What is the postfix order of the slicing tree below?



Postfix expression?

Polish Expression

- **Succinct representation of slicing floorplan**
  - Roughly specifying relative positions of blocks
- **Postorder traversal of slicing tree**
  1. Postorder traversal of left sub-tree
  2. Postorder traversal of right sub-tree
  3. The label of the current root
- **For *n* blocks, a Polish Expression contains *n* operands (blocks) and *n-1* operators (H, V)**
- **However, for a given slicing floorplan, the corresponding slicing tree (and hence polish expression) is not unique. Therefore, there is some redundancy in the representation**

Verify a Postfix Expression

- **How do we tell if a given postfix expression is valid?**
  - 12VH3: invalid
  - 123VH: valid
  - 1234567HHHHVV: valid
  - 1HVVHHV743526: invalid

Floorplan Optimization

- **Chain: HVHVH… or VHVHV…**

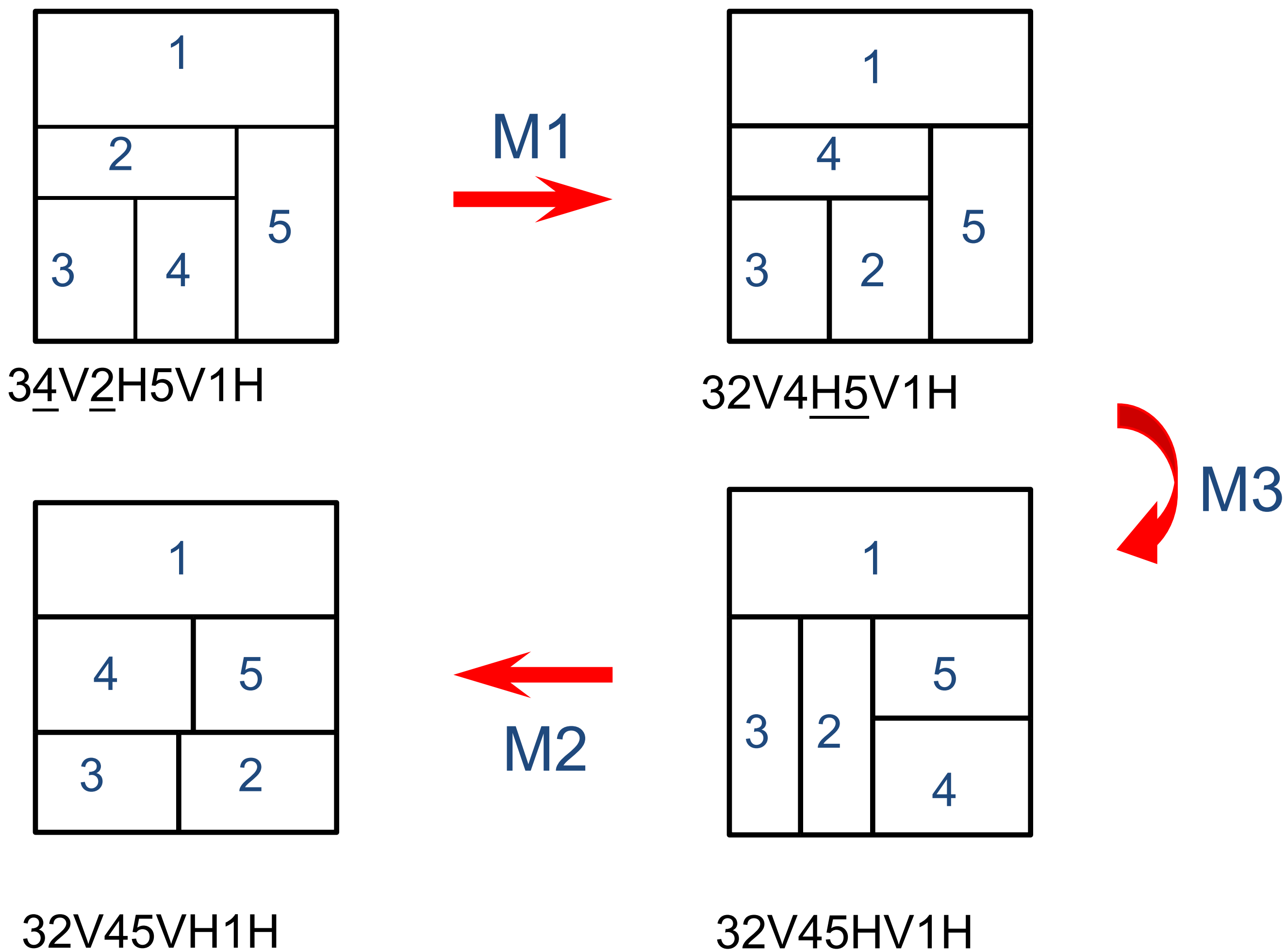
16H35V2HV74HV

Chains

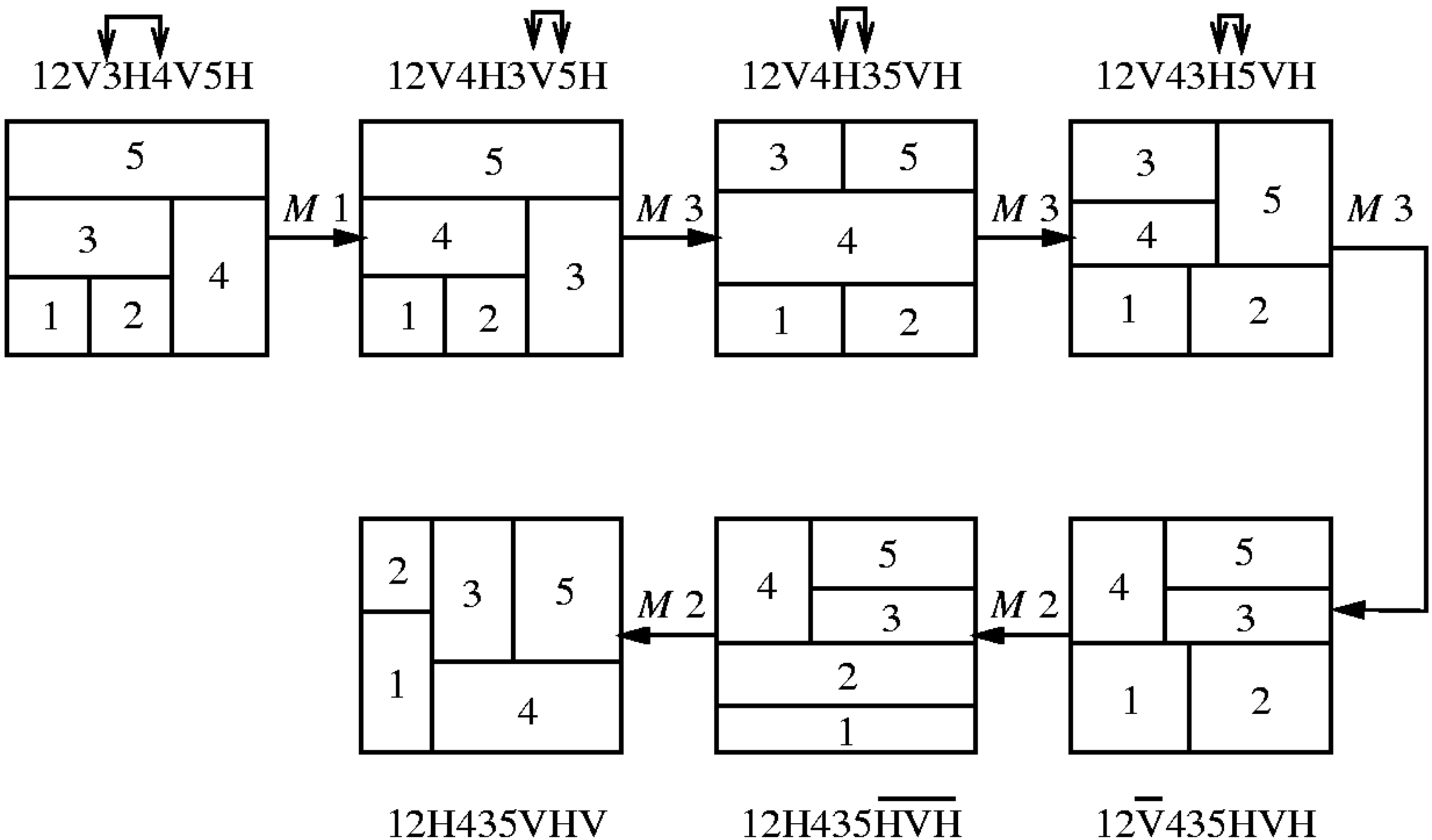
- **The moves:**
  - M1: Swap adjacent operands (ignoring chains)
  - M2: Complement some chain
  - M3: Swap 2 adjacent operand and operator
    - M3 can give you some invalid NPE. Checking for validity after M3 is needed

- **It can be proved that every pair of valid NPE are connected**

Illustration

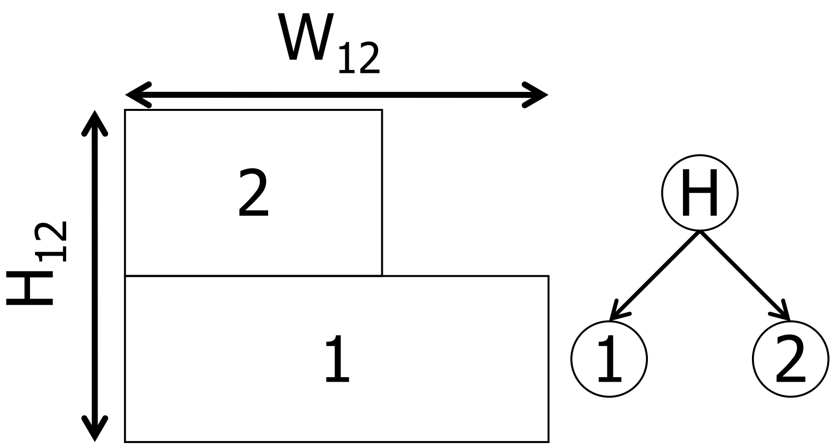


Solution Perturbation



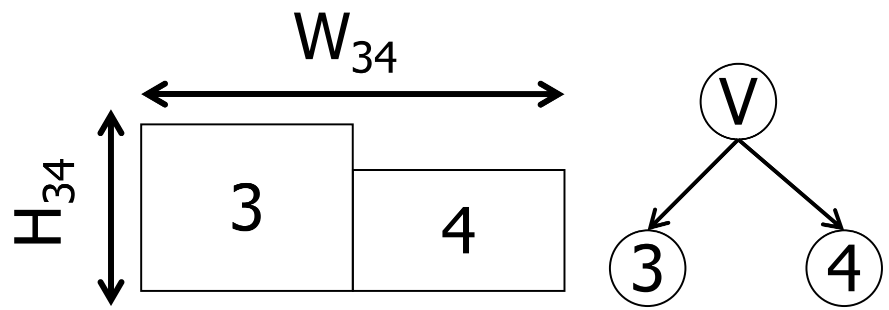
Packing from a Postfix Expression

- Binary operator
  - H: maximum on width and summation on height
  - V: maximum on height and summation on width



$W_{12} = \max(W_1, W_2)$
$H_{12} = H_1 + H_2$

(a) Postfix expression: 12H

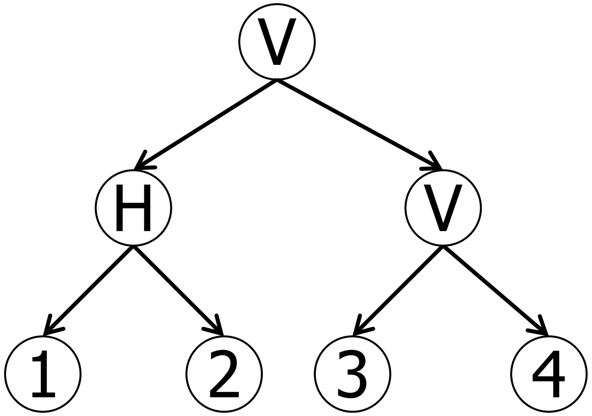


$W_{34} = W_3 + W_4$
$H_{34} = \max(H_3, H_4)$

(b) Postfix expression: 34V

Packing Two Sub-floorplans Recursively (I)

- Binary operator
  - H: maximum on width and summation on height
  - V: maximum on height and summation on width

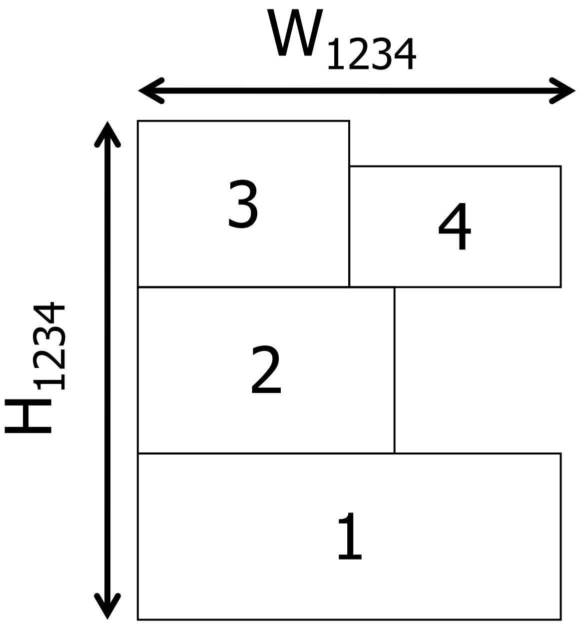


$W_{1234} = W_{12} + W_{34}$
$H_{1234} = \max(H_{12}, H_{34})$

(c) Postfix expression: 12H34VV

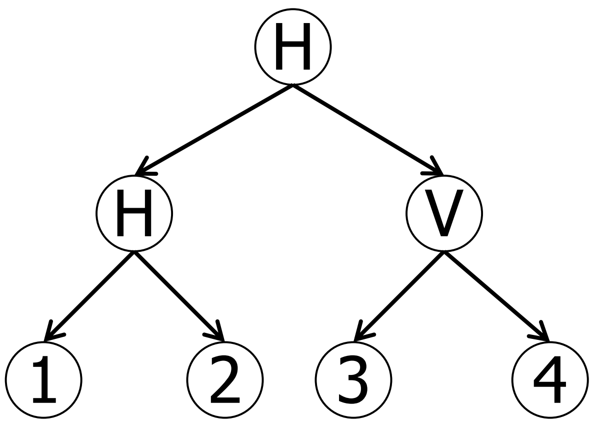
Packing Two Sub-floorplans Recursively (II)

- Binary operator
  - H: maximum on width and summation on height
  - V: maximum on height and summation on width



$W_{1234} = \max(W_{12} + W_{34})$

$H_{1234} = H_{12} + H_{34}$



(d) Postfix expression: 12H34VH

Floorplan Optimization

.Area minimization is the top priority!

.Simulated Annealing (SA)

– Randomly modify the slicing tree and select the one with the minimum floorplan area

1. Generate an initial slicing tree *T*
2. Calculate the area of the slicing tree *T*
3. Generate a random neighboring solution by changing the tree
4. Calculate the cost of the new neighboring solution
5. Compare them:
   if new_area < old_area, then move to the new solution
   else accept the new solution with a user-defined probability
6. Repeat steps 3-5 above until an acceptable solution is found

– We have provided you this optimization engine

Summary

- **Stack is a first-in-last-out data collection**
  - pop: delete an item from the stack
  - push: insert an item into the stack
  - top: query the top item in the stack
- **Queue is a first-in-first-out data collection**
  - pop: delete an item from the queue
  - push: insert an item into the queue
  - front: query the front item in the queue
- **Applications on compile expression parsing, VLSI design automation, and floorplan**