Persistent Data Structures
Ephemeral data structure

- Query operations do not change the structure.
- Update operations destroy the old version.

```
S1

3,5,8

Search(4,S1)
```
Ephemeral data structure

- Query operations do not change the structure.
- Update operations destroy the old version.

```
S1
3,5,8

S2
3,4,5,8
```

```
Insert(4,S1)
```
Ephemeral data structure

- Query operations do not change the structure.
- Update operations destroy the old version.

S1

3,5,8

Insert(4,S1)

S2

3,4,5,8

Delete(3,S2)

S3

4,5,8
Ephemeral data structure

- Query operations do not change the structure.
- Update operations destroy the old version.

```
S1
3,5,8

Insert(4,S1)
S2
3,4,5,8

Delete(3,S2)
S3
4,5,8

Search(4,S3)
```
Partially persistent data structure

Query operations can be performed on any version of the data structure.

S1

Insert(4,S1)

3,5,8

S2

Delete(3,S2)

3,4,5,8

Search(4,S2)

S3

4,5,8

Persistent Data Structures
Fully persistent data structure

- Query and update operations can be performed on any version of the data structure.

S1

3,5,8

Insert(4,S1)

S2

3,4,5,8

Delete(3,S2)

S3

4,5,8

Persistent Data Structures
Fully persistent data structure

Query and update operations can be performed on any version of the data structure.

- **S1**: 3,5,8
- **Insert(4,S1)**: 3,4,5,8
- **S2**: 3,4,5,8
- **Delete(3,S2)**: 4,5,8
- **S3**: 4,5,8
- **Insert(7,S1)**: 3,5,7,8
- **S4**: 3,5,7,8
Query and update operations can be performed on any version of the data structure.

- **S1**: 3,5,8
- **S2**: 3,4,5,8
- **S3**: 4,5,8
- **S4**: 3,5,7,8
- **S5**: 1,3,5,7,8
Fully persistent data structure

Query and update operations can be performed on any version of the data structure.

S1
3,5,8

S2
3,4,5,8

S3
4,5,8

S4
3,5,7,8

S5
1,3,5,7,8

S6
3,5,7
Confluent persistent data structure

- Full persistence, and there is also a merge operation on two or more versions.

S1
3,5,8

Insert(4,S1) → S2
3,4,5,8

S3
3,5,7,8

Insert(7,S1) → S4
1,3,5,7,8

Insert(1,S4) → S5
3,5,7

Delete(8,S3)

Persistent Data Structures
Confluent persistent data structure

- Full persistence, and there is also a merge operation on two or more versions.

S1
3,5,8

S2
Insert(4, S1)
3,4,5,8

S3
Insert(7, S1)
3,5,7,8

S4
Insert(1, S4)
1,3,5,7,8

S5
Delete(8, S3)
3,4,5,7,8

S6
3,4,5,7,8

Persistent Data Structures
Goal

Our goal is to transform an ephemeral data structures (e.g. AVL tree) into a (partially) persistent data structures.
Our goal is to transform an ephemeral data structures (e.g. AVL tree) into a (partially) persistent data structures.

Naive solution: copy the entire structure after each update.
Our goal is to transform an ephemeral data structures (e.g. AVL tree) into a (partially) persistent data structures.
Naive solution: copy the entire structure after each update.
Our goal is to transform an ephemeral data structures (e.g. AVL tree) into a (partially) persistent data structures. Naive solution: copy the entire structure after each update.

The time complexities of queries are maintained, but time complexities of updates increase to $\Theta(n)$. 

```
Insert(7,S1)
Delete(5,S2)
```
Our goal is to transform an ephemeral data structures (e.g. AVL tree) into a (partially) persistent data structures. Naive solution: copy the entire structure after each update.

The time complexities of queries are maintained, but time complexities of updates increase to $\Theta(n)$. 
**Making data structure persistent**

**Definition**

A **pointer based data structure** is a data structure composed of nodes. Each node has constant number of **pointers** and **information fields**. Additionally, there are constant number of **access pointers** (pointers to some nodes of the structure).

**Example**

In an AVL tree, every node has two pointers (left/right children), and two information field (the item, and the height balance of the node). There is one access pointer (a pointer to the root). We can optionally use more access pointers, for example pointer to the minimum element.

We will see general methods for transforming a pointer based ephemeral data structure into a persistent data structure.
Fat node method

- Every node field stores all versions of its value, each tagged with a version number. These values are stored in a balanced search tree.
- During a query on version $i$ of the data structure, the relevant value of a field is the one tagged with the largest version number less than or equal to $i$. 

![Diagram showing ephemeral and persistent data structures]

- Ephemeral
- Persistent
Fat node method

- Every node field stores all versions of its value, each tagged with a version number. These values are stored in a balanced search tree.
- During a query on version \( i \) of the data structure, the relevant value of a field is the one tagged with the largest version number less than or equal to \( i \).
Fat node method

- Every node field stores all versions of its value, each tagged with a version number. These values are stored in a balanced search tree.
- During a query on version $i$ of the data structure, the relevant value of a field is the one tagged with the largest version number less than or equal to $i$. 

\[ \text{Search}(c, S_6) \]
Fat node method

- If the update operation that creates version $i$ changes field $f$ at node $v$, add the new value to the search tree of $v$, tagged with $i$.
- If $f$ is changed several times during the update operation, store only the last value.
Fat node method

- If the update operation that creates version $i$ changes field $f$ at node $v$, add the new value to the search tree of $v$, tagged with $i$.
- If $f$ is changed several times during the update operation, store only the last value.

```
3 5
3 6 7 3 8
a
7
c
9
9
c
```

Insert($d, S_8$)

Persistent Data Structures
Fat node method

If the update operation that creates version $i$ changes field $f$ at node $v$, add the new value to the search tree of $v$, tagged with $i$. If $f$ is changed several times during the update operation, store only the last value.

```
3 5
3 6 7 3 8
a
7
b
9
9
b
```

$\text{Insert}(d, S_8)$
Fat node method

For each access pointer, store in an array the value of the pointer in every version of the data structure.
Example: AVL tree

1 2 3 4

Diagram of an AVL tree with nodes 1, 2, 3, and 4.
Example: AVL tree

1 2 3 4

key  5 1 0 balance
left 1 N
right 1 N

Insert(5)
Example: AVL tree

Insert(7)
Example: AVL tree

Insert(8)
Example: AVL tree
Example: AVL tree

Insert(6)
Total space: $O(r)$, where $r$ is the number of update steps on the ephemeral data structure.

Time: $O(\log m)$ slowdown for each operation, where $m$ is the number of versions.

For example, applying the fat node method on AVL tree gives partially persistent dictionary with $O(\log m \cdot \log n)$ time for every operation.
Node copying is another method to make ephemeral data structure partially persistent. This method requires the data structure to have bounded in-degree.

We will first handle the case of rooted trees which is simpler. Additional assumptions:

1. Either the ephemeral data structure has parent pointers, or it has only one access pointer (to the root of the tree).
2. In the ephemeral data structure, an update step on a node is performed after the update steps on its descendents.
Node copying — rooted trees

- In each node store one extra field, tagged with a version number and a field name.
- When the ephemeral update changes a field, if the extra field is empty use it to store the new value.
- Otherwise copy the node, using the latest versions of the fields.

Try to store pointer to the new copy in its parent. If the extra field at the parent is occupied copy the parent and continue.
Node copying — rooted trees

- In each node store one **extra field**, tagged with a version number and a field name.
- When the ephemeral update changes a field, if the extra field is empty use it to store the new value.
- Otherwise copy the node, using the latest versions of the fields.
  Try to store pointer to the new copy in its parent. If the extra field at the parent is occupied copy the parent and continue.

```
Insert(2, S6)
```

- Before: 
  - Node: 5
- After: 
  - Node: 5, 7, R
In each node store one extra field, tagged with a version number and a field name.

When the ephemeral update changes a field, if the extra field is empty use it to store the new value.

Otherwise copy the node, using the latest versions of the fields.

Try to store pointer to the new copy in its parent. If the extra field at the parent is occupied copy the parent and continue.
Example: binary search tree

1 2 3 4 5 6
Example: binary search tree

Insert(5)
Example: binary search tree

Insert(7)
Example: binary search tree

Insert(6)
Example: binary search tree

Insert(8)
Example: binary search tree

Insert(8)
Example: binary search tree

Delete(8)
Example: binary search tree

1 2 3 4 5 6

Insert(9)
Example: binary search tree
Slowdown for queries operation is $O(1)$.

What about update operations?

We want to bound the number of nodes copied during $m$ update operations (starting from an empty data structure).
The last copy of the node will be called live.

A used extra field of a live node holds $1.

Each update operations on the ephemeral data structure brings $2.

There are 3 types of node updates in the persistent data structure:

- Case 1: a new node due to a new node in the ephemeral data structure. $1 from the $2 amount is used to pay for the operation.

- Case 2: Change of a node field that does not induce node copying. $1 from the $2 amount is used to pay for the operation, and the remaining $1 is placed in the extra field.

- Case 3: Change of a node field that induces node copying.
Case 3

- Use the dollars in the extra fields of the copied nodes to pay for the copying.
- $1 from the $2 is used if an extra field is used.
Total space: $O(r)$, where $r$ is the number of update steps on the ephemeral data structure.

Time: $O(1)$ slowdown for query (worst case). $O(1)$ slowdown for update operation (amortized).
What is the difference between rooted trees and the general case?

In a rooted tree with our assumption, if a node \( v \) is copied, we know its parent \( w \) so we can easily update the pointer to \( v \) in \( w \).

In the general case, when copying \( v \) we need to update all pointers to \( v \), so we need to know which nodes contain a pointer to \( v \).
Node copying — general case

- Let $p$ an upper bound on the in-degrees of the nodes.
- Each persistent node stores:
  - $p$ extra fields
  - $p$ back pointers (if a live node $x$ has a pointer to $y$, store a back pointer to $x$ in $y$)
  - 1 copy pointer (points to the next copy of the node)
  - Time-stamp (version in which it was created)

Simulation of an update operation on the ephemeral data structure is done in two stages:

- Simulate each update step, and copy nodes. When $x$ is copied to $c(x)$, add $x$ to a queue $S$.
- While $S$ is not empty, remove a node $x$ from $S$. Using the back pointers, update pointers to $x$ to point to $c(x)$. If a node is copied, add it to $S$. 

Persistent Data Structures
Suppose $x$ is copied (in the first or second stage), and let $c(x)$ be the new copy.
In the example below, $y_1$ was also copied, but the pointer to $y_1$ in $x$ wasn’t updated.
If the latest version of a pointer field $f$ in $x$ points to $y$, change field $f$ in $c(x)$ to point to $c(y)$ if it exists, and to $y$ otherwise. Update the back pointers pointing to $x$. 

Diagram:

- **Ephemeral**
- **Back**
- **Copy**
When $x$ is removed from $S$, for every back pointer to $z$, find in $z$ the pointer to $x$. If its version is $i$, change it to $c(x)$. Otherwise, add a pointer to $c(x)$ with time-stamp $i$ to the extra fields of $z$. Update the back pointer.
If all the extra fields of $z$ are used, copy $z$ and add $z$ to $S$. 

\[
\begin{array}{c}
\text{ephemeral} \\
\text{back} \\
\text{copy}
\end{array}
\]
Partial persistence — summary

Fat node method:
- Total space: $O(r)$, where $r$ is the number of update steps on the ephemeral data structure.
- Time: $O(\log m)$ slowdown for each operation, where $m$ is the number of versions.

Node copying method:
- Total space: $O(r)$.
- Time: $O(1)$ slowdown for query (worst case).
  $O(1)$ slowdown for update operation (amortized).
Applications — planar point location

A planar point location data structure stores a partition of the plane into polygons, and allows the following queries: given a point \( p \), find the polygon containing \( p \).
Applications — planar point location

A planar point location data structure stores a partition of the plane into polygons, and allows the following queries: given a point $p$, find the polygon containing $p$. 

![Diagram of a planar point location data structure](image-url)

- [Diagram of a planar point location data structure](image-url)
A planar point location data structure stores a partition of the plane into polygons, and allows the following queries: given a point $p$, find the polygon containing $p$. 
Applications — planar point location

Partition the plane into vertical slabs by drawing a vertical line through each endpoint.
Applications — planar point location

Store the $x$-coordinates of segments endpoints in a static successor data structure (for example, sorted array).
For each slab, store the segments contained in the slab in a binary search tree, sorted according to their vertical order.
To answer a query $p = (x, y)$, find the slab containing $p$. Then, find the highest segment in the slab which is below $p$. 
Applications — planar point location

- Query time: $\Theta(\log n)$.
- Space complexity: $\Theta(n^2)$.
- Preprocessing time: $\Theta(n^2)$. 

![Diagram of a planar point location problem](image)
Applications — planar point location

1,2 → 1,3,4,5 → 6,7,3,4,5

Delete 2
Insert 3,4,5
Delete 1
Insert 6,7
Instead of storing a search tree for each slab, use a partially persistent AVL tree.

\[
\begin{array}{ccc}
1,2 & \rightarrow & 1,3,4,5 \\
\text{Delete 2} & \text{Delete 1} & \\
\text{Insert 3,4,5} & \text{Insert 6,7} & 6,7,3,4,5
\end{array}
\]

\(m = \text{number of update operation} = 2n\).
\(r = \text{total number of node updates in the ephemeral AVL tree} = O(n \log n)\).

Using fat node method:

- Query time: \(O(\log^2 n)\).
- Space complexity: \(O(n \log n)\).
- Preprocessing time: \(O(n \log^2 n)\).
Applications — planar point location

Instead of storing a search tree for each slab, use a partially persistent AVL tree.

\[
\begin{array}{c}
1,2 \\
\text{Delete 2}
\end{array} \quad \begin{array}{c}
1,3,4,5 \\
\text{Insert 3,4,5}
\end{array} \quad \begin{array}{c}
6,7,3,4,5 \\
\text{Delete 1} \quad \text{Insert 6,7}
\end{array}
\]

\(m = \text{number of update operation} = 2n.\)

\(r = \text{total number of node updates in the ephemeral AVL tree} = O(n \log n).\)

Using node copying method:

- Query time: \(O(\log n).\)
- Space complexity: \(O(n \log n).\)
- Preprocessing time: \(O(n \log n).\)